FVPP: Simboličko predstavljanje stanja BDD-ovima

Algoritmi u pozadini alata za provjeru modela

Pripremio: izv. prof. dr. sc. Alan Jović

Ak. god. 2022./2023.

Sadržaj

- Predstavljanje Booleovih funkcija
- Binarni dijagrami odlučivanja
- Algoritam ITE i implementacija BDD-ova
- Primjena BDD-ova

PREDSTAVLJANJE BOOLEOVIH FUNKCIJA

Simbolički postupci verifikacije sustava

- Rješenje problema eksplicitnih postupaka verifikacije provjerom modela:
  - Eksplozija stanja čini neefikasnim manipulaciju pojedinačnim stanjima u memoriji računala.
  - Potrebno je uvesti nove načine predstavljanja velikih skupova stanja u memoriji računala.
  - **Simbolički postupci** temelje se na uporabi **logičkih (Booleovih) funkcija**.
  - Simbolički postupci manipuliraju skupovima stanja a ne individualnim stanjima i time smanjuju potrebu za resursima.

Temeljne ideje u simboličkom pristupu

- Skupovi (relacije) kodiraju se Booleovim funkcijama.
- Booleove funkcije potrebno je prikazati i zatim njima upravljati u računalu na najučinkovitiji način (s aspekta prostornih i vremenskih zahtjeva).
- BDD dijagrami su vrlo učinkovita metoda predstavljanja Booleovih funkcija.
- Sve operacije nad skupovima stanja (izračun čvrste točke, dosezljivost, logičke operacije, itd.) obavljaju se s BDD dijagramima.

Predstavljanje skupova i relacija Booleovim funkcijama

- Skupove s konačnim brojem elemenata i binarne relacije nad tim skupovima moguće je prikazati Booleovim funkcijama i to najprije kodiranjem elemenata skupova odnosno relacija

- Neka je zadan skup stanja $\mathbf{S} = \{s_1, s_2, \ldots, s_m\}$
- m elemenata skupa možemo binarno kodirati pomoću n bitova, $n = \lceil \log_2 (m) \rceil$ – najmanji cijeli broj veći od $\log_2 (m)$

- Binarnu relaciju $\mathbf{R} \subseteq \mathbf{S} \times \mathbf{T}$ također možemo kodirati.

Predstavljanje skupova i relacija Booleovim funkcijama

Primjer:

- S = {[0, 3], [8, 15], [16, 23]} – podskup skupa cijelih brojeva
- 20 elemenata, kodiranje s 5 binarnih varijabli: (x1, x2, x3, x4, x5)

  Prvi element (0): 00000

  Zadnji element (23): 10111

- Kodiranje (x1 teži 2^5, a x5 teži 2^0):

x1	x2	x3	x4	x5	f_S	elementi	broj elemenata (ukupno 20)	
0	0	0	-	-	1	[0 - 3]	4	(x4, x5 sve kombinacije)
0	1	-	-	-	1	[8 - 15]	8	(x3, x4, x5 sve kombinacije)
1	0	-	-	-	1	[16 - 23]	8	(x3, x4, x5 sve kombinacije)

- **Karakteristična funkcija pripadnosti** elemenata skupa S: $f_S : \{0, 1\}^n \rightarrow \{0, 1\}$

  $f_S = x1'\ x2'\ x3' + x1'\ x2 + x1\ x2'$ (x3, x4, x5 u minimizaciji nestaju)

- 20 elemenata skupa S predstavljeno je sa samo 3 binarne varijable.

  Kodiranje se isplati !!!

- Tipično: broj binarnih varijabli kodiranja ≈ log(| S |)

Predstavljanje skupova i relacija Booleovim funkcijama

Primjer:

- S = {2, 4, 5, 7}
- T = {3, 5, 8}

1. Naći njihove karakteristične funkcije uz kodiranje:

- (x_i, y_k - težinske binarne varijable kodiranja)

s_i	x_1	x_2
2	0	0
4	0	1
5	1	0
7	1	1

t_i	y_1	y_2
3	0	0
5	0	1
8	1	0

- **f_S = 1** (!!!) f_T = y1' y2' + y1' y2 + y1 y2' **= y1' + y2'** (minimizirano)

Predstavljanje skupova i relacija Booleovim funkcijama

2. <u>Naći karakterističnu funkciju relacije:</u>

- $f_R = \{(s, t) \in (S \times T) \mid t = s + 1\}$

- Iz kartezijskog produkta izaberemo podskup parova koji zadovoljavaju zadanu relaciju. Slijedi podskup:
  $S = \{2, 4, 5, 7\}$
  $T = \{3, 5, 8\}$
  $R = \{(2, 3), (4, 5), (7, 8)\}$

- Uz ranije zadano kodiranje:

	x1	x2	y1	y2
(2, 3)	0	0	0	0
(4, 5)	0	1	0	1
(7, 8)	1	1	1	0

- Karakteristična funkcija relacije R je:

$$f_R = x1'\, x2'\, y1'\, y2' + x1'\, x2\, y1'\, y2 + x1\, x2\, y1\, y2'$$

Predstavljanje Booleovih funkcija

- Kodiranje skupova i relacija binarnim varijablama traži **učinkovito predstavljanje logičkih (Booleovih) funkcija f(*x*) u računalu**

  $f(x) : B^n \rightarrow B,\ \ B = \{0, 1\}, x = \{x_1, x_2, \ldots, x_n\}$

- **Kako prikazati Booleovu funkciju u računalu? – SVI OVDJE NAVEDENI PRISTUPI SU NEUČINKOVITI:**

1. Predstavljanje logičkih funkcija **tablicom istinitosti**
2. Predstavljanje logičkih funkcija **mintermima i makstermima**
3. Predstavljanje logičkih funkcija u standardnom (dvorazinskom) obliku **sume produkata (SOP)** ili **produkta suma (POS)**
4. Predstavljanje logičkih funkcija u **nestandardnom** (višerazinskom) obliku
5. Predstavljanje logičkih funkcija **Booleovim kockama,** odnosno **Karnaughovim (K) tablicama**

Predstavljanje Booleovih funkcija

1. **Tablica istinitosti**
   - Prednost: kanonski prikaz (jedinstven, svaka varijabla se pojavljuje), razumljiv, jednostavna provjera ekvivalencije.
   - Nedostatak: eksponencijalni prostor za pohranu (za *n* varijabli 2^n redaka – interpretacija u tablici).

2. **Suma mintermi ili produkt makstermi**
   - Minterm (npr. m_6 = x y z') ili maksterm (npr. M_6 = x'+y'+z) sadrže sve varijable, prirodno se dobivaju iz tablice istinitosti
   - Prednost: kanonski prikaz, razumljiv, jednostavna provjera ekvivalencije.
   - Nedostatak: nije minimalan oblik funkcije, a time se gubi na optimalnosti prikaza.

Predstavljanje Booleovih funkcija

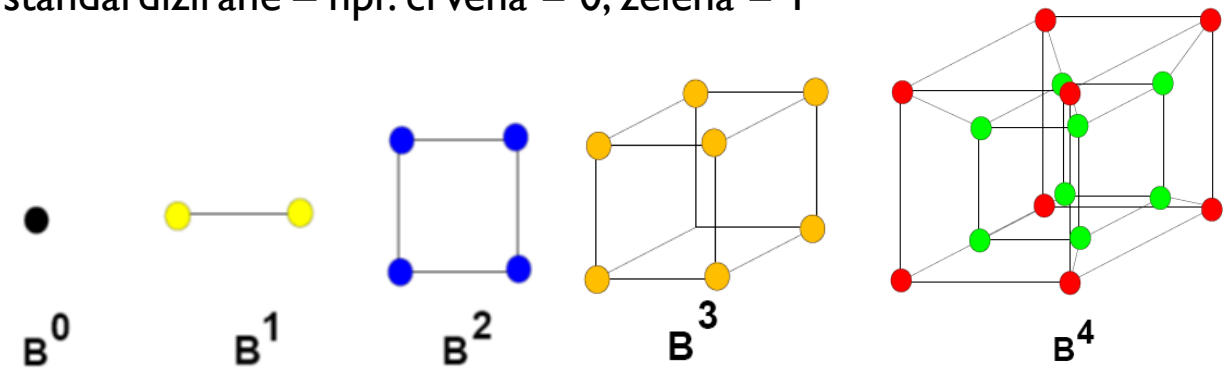
3. **Suma produkata (SOP) ili produkt suma (POS)**
   - SOP – npr. $f = x_1 x_2' + x_1' x_3$
   - POS – npr. $f = (x + y)(z + w)$
   - Konverzija kanonskih oblika (sume mintermi ili produkta makstermi) u SOP ili POS radi se primjenom aksioma i teorema Booleove algebre
   - Prednost: kompaktan opis
   - Nedostatak: nije kanonski, nije nužno minimalan, konverzija SOP $\Leftrightarrow$ POS vrlo skupa, skupo određivanje ekvivalencije i zadovoljivosti

4. **Nestandardni (višerazinski opis)**
   - Npr. $f = (xy + uv)(x'y' + u'w')$
   - Prednost: vrlo kompaktan opis
   - Nedostatak: nije kanonski, računalno vrlo skupa pretvorba u kanonske oblike, skupo određivanje ekvivalencije i zadovoljivosti

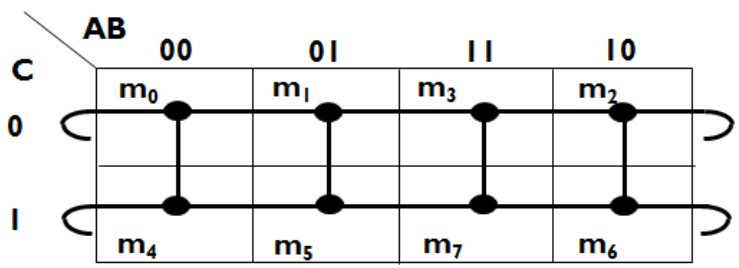
Booleove kocke i *K*-tablice

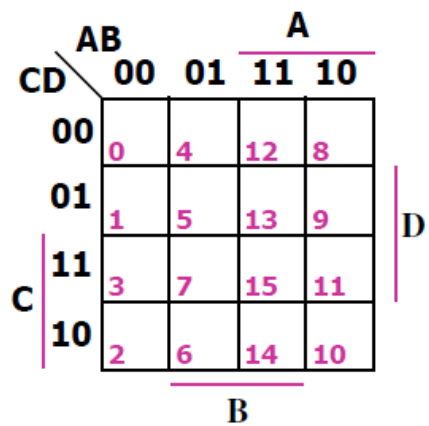
5. **Booleova kocka** je geometrijsko tijelo koje predstavlja Booleovu funkciju. Svaki vrh je jedan minterm funkcije, a vrh je obojen sukladno vrijednosti funkcije (0 ili 1) za taj minterm

   Boje <u>nisu</u> standardizirane – npr. crvena = 0, zelena = 1



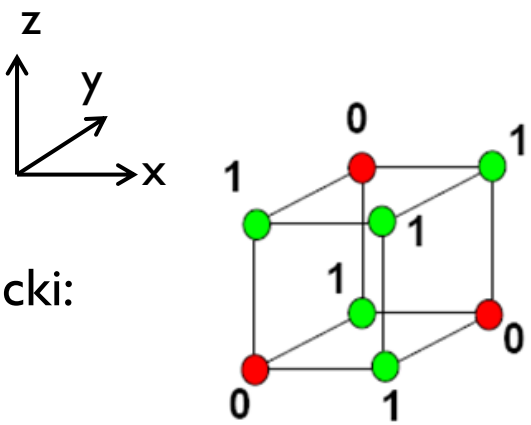
Dvodimenzionalni prikaz Booleove kocke = **Karnaughova (K) tablica**:





Booleove kocke

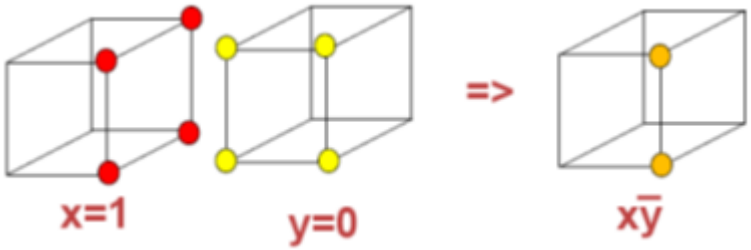
- Svaka kombinacija vrhova Booleove kocke je zasebna logička funkcija:



npr.

f = xy'z' + x'y'z + x'yz' + xy'z + xyz odgovara kocki:

- Operacije nad literalima se također mogu smatrati operacijama nad Booleovim kockama:
- Npr. *C* = *x y'*

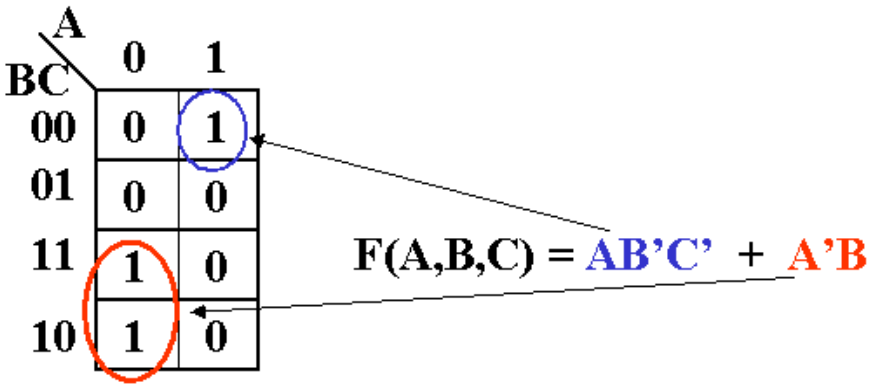


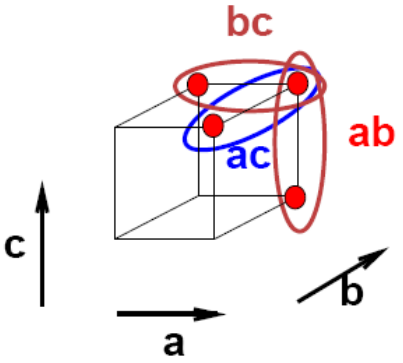
Booleove kocke

- Predstavljanje funkcije kao SOP može se zamisliti kao skup Booleovih kocki

  npr. $f = ab + ac + bc$, $f = \{ab, ac, bc\} = C$

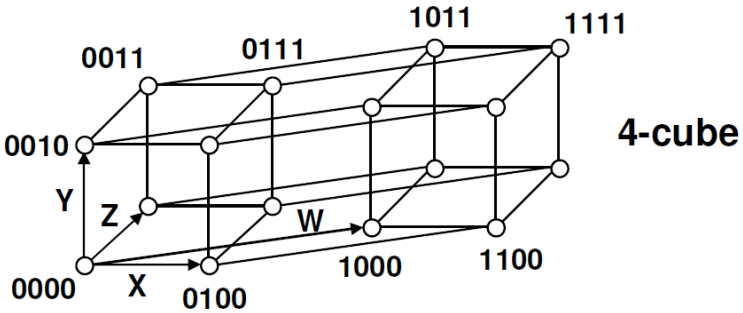
- Skup Booleovih kocki koje predstavljaju logičku funkciju naziva se **pokrivanje** (engl. *cover*, *C*) funkcije *f*
- Svaki vrh s vrijednosti funkcije 1 pokriven je barem jednom kockom
- Postupak dvorazinske minimizacije traži najmanji broj kocki pokrivanja (slično kao i minimizacija *K*-tablicom u 2D prikazu)





Booleove kocke

- Prednosti: kanonski, jednostavna provjera ekvivalencije, vizualno razumljivo predstavljanje logičkih funkcija
- Nedostatak: za n > 4 predstavljanje postaje neprimjereno



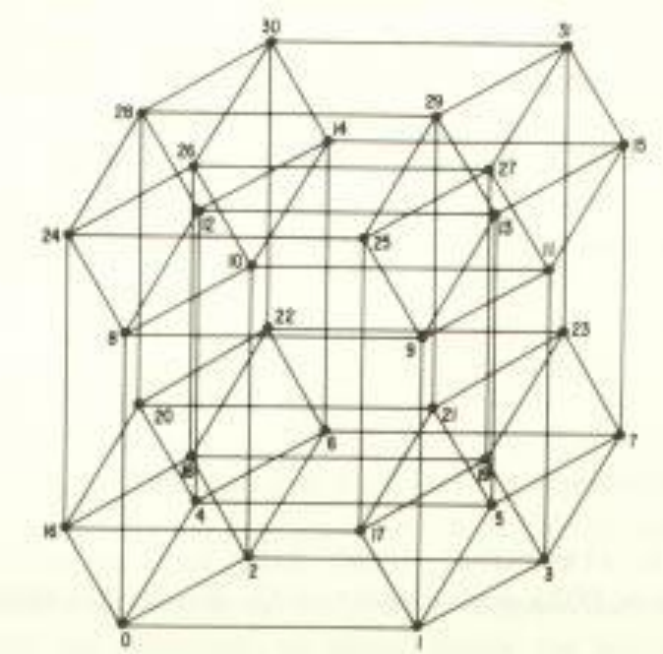


Fig. 1.15 Frisiani's rendition of 5-cube

Primjena predstavljanja Booleovih funkcija: analiza dosezljivosti

- U analizi dosezljivosti zanima nas **skup stanja dostupan iz početnih stanja Q_0 u jednom ili više koraka**. Na početku definiramo funkcije koje će nam dati skup stanja dosezljiv u **jednom koraku** (sljedeća stanja, engl. *next state*).
- Neka je dan **FSM = (Q, Σ, Δ, δ, Q_0, γ),** gdje je Q – skup stanja, Σ – skup ulaznih vrijednosti, Δ – skup izlaznih vrijednosti, δ – funkcija sljedećeg stanja (δ za neki ulaz $x \in \Sigma$ i stanje $s \in Q$ daje jedno sljedeće stanje $t \in Q$), Q_0 – skup početnih stanja, γ – funkcija izlaza.

- Definiramo pomoćnu **logičku** funkciju malo "eta" $\eta(s, t)$:

  $\eta(s, t) = 1$ **akko** $\exists_{x=\Sigma} \mid \delta(s, x) = t$

  $\eta(s, t) = 0$ **inače** (postoji li ili ne $(s, x, t) \in \delta$)
- Zatim definiramo funkciju H (veliki "eta") koja će dati **skup stanja** T **{t∈Q} dostupnih u jednom koraku** iz skupa zadanih stanja $S \subseteq Q$:
- $H: 2^Q \rightarrow 2^Q$, H je funkcija nad svim podskupovima od **Q**, također $H(\emptyset) = \emptyset$.

$H(S) = \{t \in Q \mid \exists_{s \in S}\ \eta(s, t)\}$ **= sva sljedeća stanja**

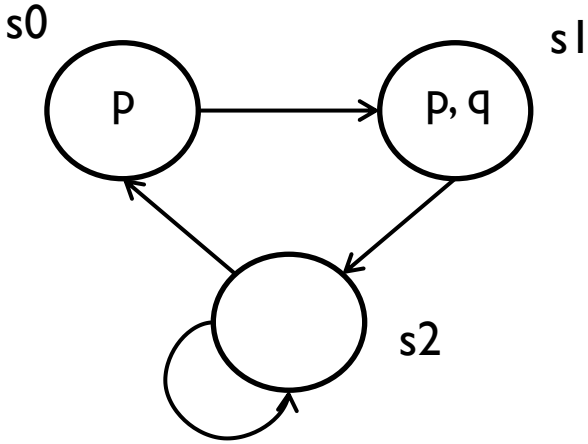
Kodiranje FSM-a

- Uobičajene oznake za $FSM = (Q, \Sigma, \Delta, \delta, Q_0, \gamma)$ preslikavaju se u oznake koje označuju **kodirane skupove**.
- $FSM_{KODIRANO} = (S, I, O, N, S_0, Y)$

- Uz $B = \{0, 1\}$, kodiranje daje:
- $S = B^n$ skup stanja (n = broj bitova za kodiranje)
- $I = B^m$ skup ulaznih vektora od m bitova
- $O = B^p$ skup izlaznih vektora od p bitova
- $N : B^n \times B^m \rightarrow B^n$ funkcija sljedećeg stanja
- $S_0 \subseteq S$ skup početnih stanja
- $Y : B^n \times B^m \rightarrow B^p$ funkcija izlaza

- **Time se sve značajke FSM-a mogu izraziti preko Booleovih logičkih funkcija**

Primjer kodiranja Kripkeove strukture M(S,R,L)



S = {s0, s1, s2}

R = {(s0,s1),(s1,s2), (s2,s2), (s2,s0)}

L(s0) = {p}

L(s1) = {p,q}

L(s2) = {}

S_{enc} = {(1,0), (1,1), (0,0)}

R_{enc} = {(1,0,1,1), (1,1,0,0),(0,0,0,0), (0,0,1,0)}

f_{Senc}=pq' + pq + p'q'

f_{Renc}=p1q1'p2q2 + p1q1p2'q2' + p1'q1'p2'q2' + p1'q1'p2q2'

Rješenje

- Potrebno je uvesti novu matematičku strukturu koja će omogućiti učinkoviti zapis karakterističnih Boolevih funkcija u računalu
- Takvom strukturom omogućit će se učinkovitija analiza dosezljivosti pomoću razmatranja skupova stanja, a ne samo pojedinačnih stanja
- Za izgradnju takve strukture u računalu koristit će se posebno oblikovani algoritam

Zadatci

1. Za varijablu programa zadanu kao podskup cijelih brojeva $S = \{3..11\}$ odredite njezinu karakterističnu Booleovu funkciju

2. Naći karakterističnu funkciju relacije

   $R = \{(s, t) \in (S \times T) \mid t = s + 1\}$, pri čemu je:

   $S = \{2, 4, 6, 8\}$, $T = \{1, 5, 7, 9, 10\}$

BINARNI DIJAGRAMI ODLUČIVANJA

Što tražimo od prikaza Booleovih funkcija?

- Kanonski (jedinstveni način prikaza)
- Što manje zauzeće vremenskih i prostornih resursa u računalu
- Efikasne operacije provjere ekvivalentnosti, tautologije, zadovoljivosti formule i sl.
- Efikasne operacije verifikacije provjerom modela (dostupnost stanja, izračun "čvrste točke", i sl.)
- Sve navedene ciljeve zadovoljava predstavljanje logičkih funkcija **binarnim dijagramima odlučivanja (BDD)**

Kofaktori i teorem ekspanzije

- **Shannonovi (Booleovi) kofaktori**
  - Neka je $f(x) : B^n \rightarrow B$, $x = (x_1, \dots x_n)$ logička, Booleova funkcija
  - **Kofaktori** f_x (pozitivni) odnosno $f_{x'}$ (negativni) funkcije f po varijabli x_i :

$$f_{x_i}(x_1, \dots, x_n) = f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n)$$
$$f_{x_i'}(x_1, \dots, x_n) = f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n)$$

  - To je funkcija f u kojoj je fiksirano $x_i = 1$ $(x_i' = 0)$, odnosno $x_i = 0$ $(x_i' = 1)$.

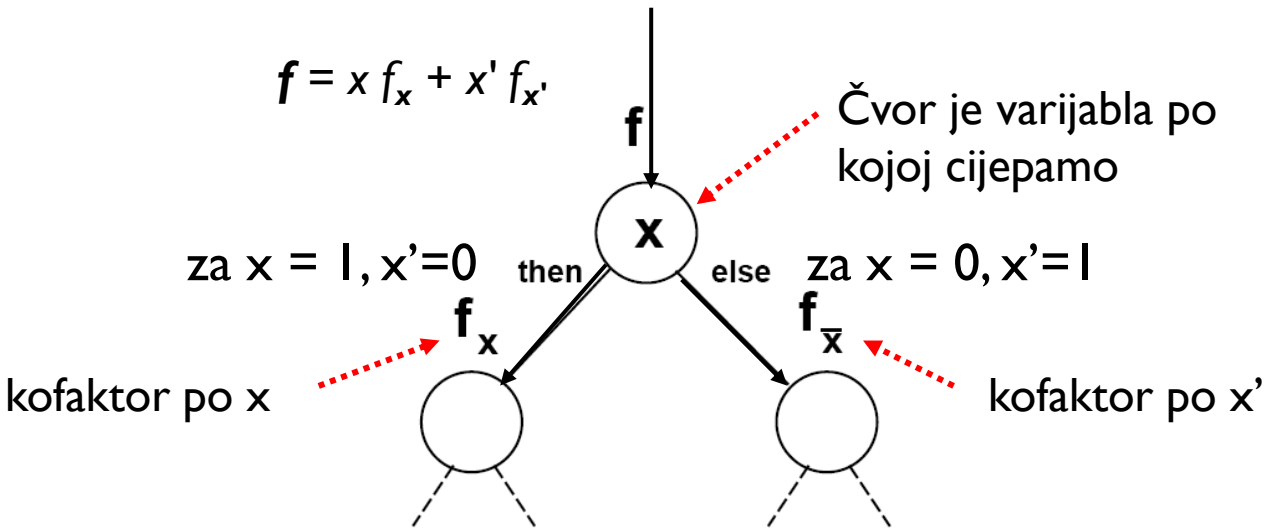
- **Shannonov teorem ekspanzije**
  - Svaka logička funkcija može se ekspandirati oko varijable x_i u SOP oblik:

$$f(x) = x f_x + x' f_{x'}$$

  - Varijabla x_i naziva se **varijabla cijepanja** (engl. *splitting variable*).
  - Pri tome su f_x i $f_{x'}$ kofaktori

Prikaz Shannonove ekspanzije grafom

- Cijepanje po varijabli *x* prikazuje se **grafom** (binarnim stablom):



Najčešća uporaba **<u>oznaka na lukovima</u>**: ***1, 0*** ili ***T, E*** (od Then, Else)

Binarni dijagram odlučivanja
(BDD – engl. *Binary Decision Diagram*)

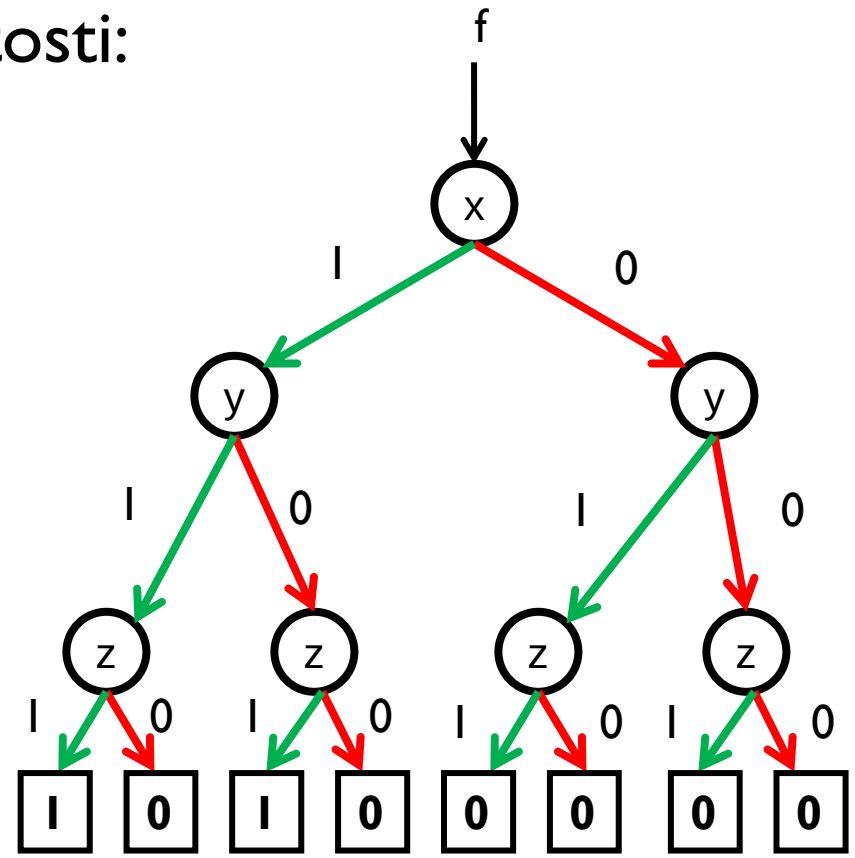
- **G(V, E)** – Usmjereni, aciklički graf (DAG, engl. *Directed Acyclic Graph*)
  - ***V*** – skup čvorova, **nezavršni** čvorovi označuju **varijable** $x_i \in X$, **završni** čvorovi označeni su **konstantama** 0 ili 1.
  - ***E*** – skup **usmjerenih** lukova $E \subseteq V \times V$, označenih **1 (then, T)** ili **0 (else, E)**.
  - *X* – **uređen skup varijabli**. *rank(v)* označuje **redoslijed** oznake čvora *v*.
  - **Svaki put u grafu slijedi zadanu uređenost varijabli.**
  - Svaki **nezavršni čvor** x_i predstavlja neku logičku funkciju koja se cijepa po toj varijabli:

    $f = x\, f_x + x'\, f_{x'}$, gdje su kofaktori: $f_x = f|_{x=1}$, $f_{x'} = f|_{x=0}$ izravni sljedbenici čvora x_i na lukovima **then** (**T**, **1**) i **else**(**E**, **0**).

BDD

- Primjer BDD-a za funkciju *f* zadanu tablicom istinitosti:

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



ROBDD – smanjeni, uređeni binarni dijagrami odlučivanja

Engl. Reduced Ordered Binary Desision Diagram

- Smanjenje i uređenje BDD-a očituje se u tome da:
  - Ne postoji čvor *v* takav da then(*v*) = else(*v*), tj. čvor nema jednaku djecu
  - Izomorfni podgrafovi (podgrafovi s istom strukturom i označavanjem) se svode na jedan podgraf.
  - Crtaju se samo dva završna čvora, 1 i 0, a ne dva završna za svaki nezavršni čvor na zadnjoj razini grafa kao kod „običnog" BDD-a

Postupak izgradnje ROBDD-a iz funkcije zadane tablicom istinitosti

1. Nacrtaj čitavi BDD, označi istinitosti u završnim čvorovima
2. Sad nacrtaj samo dva završna čvora (1 i 0), preusmjeri sve veze od zadnjih nezavršnih čvorova u odgovarajuće završne čvorove
3. Kretajući se od razine zadnjih nezavršnih čvorova prema korijenu grafa, najprije ukloni sve čvorove koji imaju **then**(v) = **else**(v), a u drugom prolazu sve izomorfne podgrafove zamijeni s jednim na istoj razini

Primjer 1: Nacrtaj ROBDD za funkciju f

uz uređenje: x < y < z:

Najprije varijabla x, pa y, pa z

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

ROBDD - kanoničnost

- ROBDD predstavljaju **kanonski (jedinstveni) oblik** logičke funkcije (Bryant, 1986.)

Posljedice kanoničnosti:

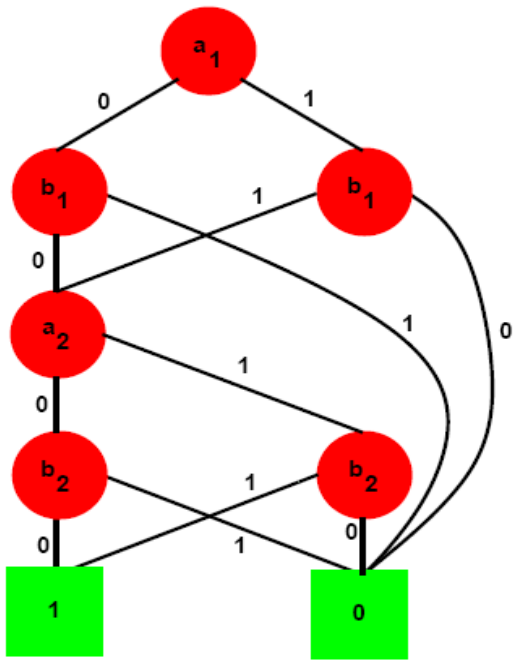
- Dvije logičke funkcije su **ekvivalentne** ako su njihovi ROBDD grafovi izomorfni. Izomorfnost je postignuta ako postoji bijekcija između grafova takva da se završni čvorovi preslikavaju u završne, a nezavršni u nezavršne s istim vrijednostima čvorova djece. Uvjet je **jednaka uređenost** varijabli.
- Završni čvor **1** predstavlja logičku funkciju **f = 1**.
- Završni čvor **0** predstavlja logičku funkciju **f = 0**.
- Odlučivanje o ekvivalentnosti logičkih formula i zadovoljivosti, koja se svodi na pokazivanje ne-ekvivalencije s grafom funkcije f = 0, su bitno pojednostavljeni.

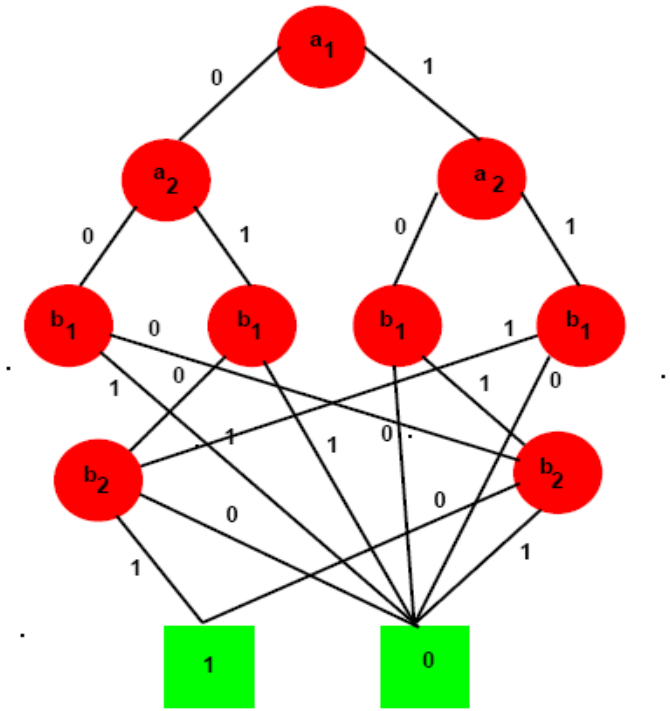
ROBDD – uređenost varijabli

Složenost ROBDD-a **značajno ovisi o uređenosti** (redoslijedu) varijabli.

Neke funkcije imaju eksponencijalno velike ROBDD-ove za svaku uređenost varijabli, dok neke imaju polinomijalno (pa i linearno) velike BDD-ove za neka uređenja, a eksponencijalno za druga. **Određivanje optimalne uređenosti – NP-težak problem.**

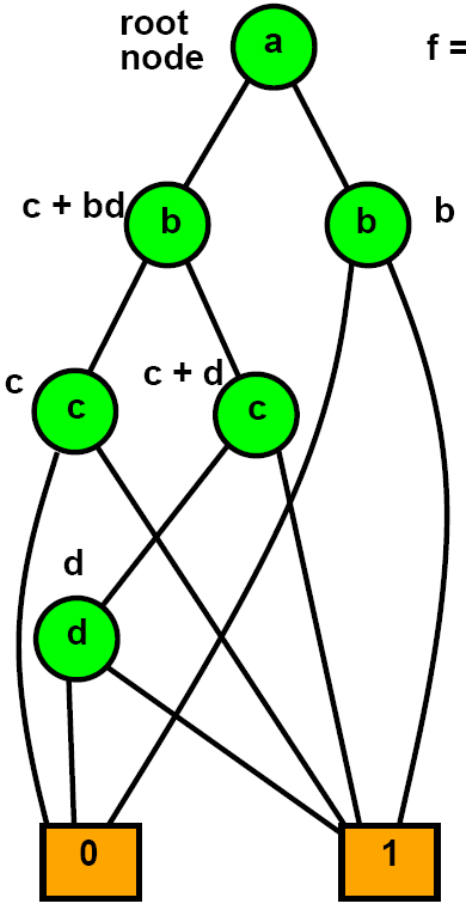
Primjer dvobitnog komparatora: Na lijevoj slici uređenost ($a_1 < b_1 < a_2 < b_2$), na desnoj slici uređenost ($a_1 < a_2 < b_1 < b_2$)





Učinkovitost prikaza funkcija BDD-ovima

- ROBDD ne numerira eksplicitno putove (kao tablica), već graf s **linearnim brojem čvorova** predstavlja **eksponencijalan broj putova**
- ROBDD-ovi se u praksi koriste kad želimo za neki problem pronaći **skup svih mogućih rješenja**
- Za razliku od ROBDD-a, SAT-rješavači (tema idućeg predavanja) se koriste samo za vrlo učinkovit pronalazak jednog rješenja (ako ono postoji)



ALGORITAM ITE I IMPLEMENTACIJA BDD-OVA

Operator ITE

- U izgradnji ROBDD-a ponavlja se postupak: odabere se varijabla iz uređenog skupa i funkcija se cijepa sukladno Shannonovom teoremu.
- Za implementaciju izgradnje ROBDD-a u računalu, potrebno je pronaći pogodnu **funkciju (operator)** za navedeni postupak i izgraditi algoritam njezine uporabe.
- Jedna takva pogodna funkcija s **tri argumenta** je Booleova funkcija (operator) ***ite***:

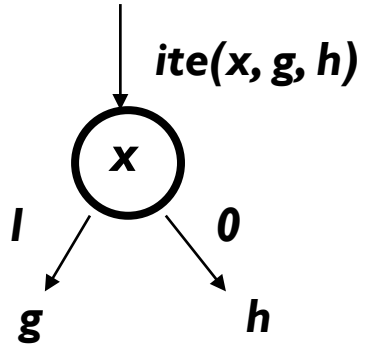
$$ite(f, g, h) = f\,g + f'\,h$$

- gdje su ***f, g, h*** proizvoljne logičke (Booleove) funkcije. Ta se funkcija interpretira (***i-t-e***):
- **I**F *f* (tj. ako *f=1*) **T**HEN *g*, **E**LSE (tj. ako *f=0*) *h*.
- *To je generalizacija sklopa multipleksor (u sklopu multipleksor uobičajeno funkcije f, g, h su pojedinačne Booleove varijable).*

Odnos ITE-a i BDD-ova

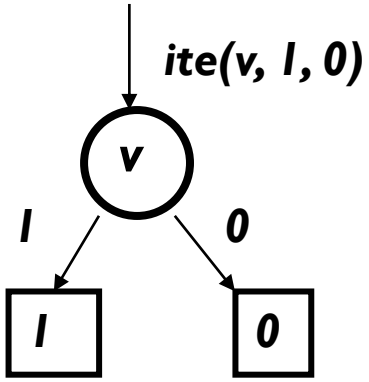
- Neka je ***x*** Booleova varijabla.
- ***f* = *x*** je jedna logička funkcija.
- Ako to uvrstimo u ***ite*** operator slijedi:

  ite(x, g, h) = x g + x' h
- Izraz predstavlja cijepanje po varijabli ***x*** (sukladno Shannonu) što možemo prikazati grafički:



- <u>Slijedi:</u> ako ***ite*** operator ima kao **prvi parametar** **samostalnu (vršnu) varijablu**, tada ***ite*** predstavlja BDD s čvorom te varijable, te lijevim lukom (1, T) koji završava u funkciji ***g***, a desnim (0, E) u funkciji ***h***.

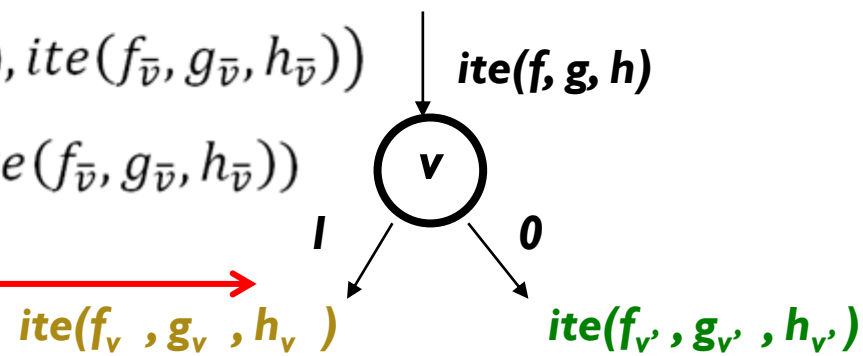
- Neka je ***v*** Booleova varijabla.
- ***f* = *v*** (**funkcija jedne varijable**) može se predstaviti operatorom ***ite*** kao: ***ite(v, 1, 0)***
- Odnosno grafički:



Rekurzija u ITE

- Neka su općenite logičke funkcije *f*, *g*, *h* zadane svaka sa svojim ROBDD-om.
- Neka je ***ite(f, g, h) = (f g + f' h)*** funkcija triju argumenata (logičkih funkcija) te neka je varijabla ***v*** prva u nizu uređenih varijabli svih triju funkcija (*f*, *g*, *h*). Rastavljanje *ite(f, g, h)* po Shannonu pokazuje **rekurziju**:

$$
\begin{aligned}
ite(f,g,h) &= fg + \bar{f}h \\
&= v(fg + \bar{f}h)_v + \bar{v}(fg + \bar{f}h)_{\bar{v}} \\
&= v(f_v g_v + \bar{f}_v h_v) + \bar{v}(f_{\bar{v}} g_{\bar{v}} + \bar{f}_{\bar{v}} \bar{h}_{\bar{v}}) \\
&= ite(v, ite(f_v, g_v, h_v), ite(f_{\bar{v}}, g_{\bar{v}}, h_{\bar{v}})) \\
&= (v, ite(f_v, g_v, h_v), ite(f_{\bar{v}}, g_{\bar{v}}, h_{\bar{v}}))
\end{aligned}
$$



To je **ROBDD** s vršnom varijablom ***v***

Ostvarivanje Booleovih operacija ITE-om

- **ITE-operator jednostavno implementira svih 16 logičkih funkcija s dvije varijable:**

Table	Subset	Expression	Equivalent Form
0000	0	0	0
0001	AND(f, g)	fg	ite(f, g, 0)
0010	f > g	$f\bar{g}$	ite(f, $\bar{g}$, 0)
0011	f	f	f
0100	f < g	$\bar{f}g$	ite(f, 0, g)
0101	g	g	g
0110	XOR(f, g)	$f \oplus g$	ite(f, $\bar{g}$, g)
0111	OR(f, g)	$f + g$	ite(f, 1, g)
1000	NOR(f, g)	$\overline{f + g}$	ite(f, 0, $\bar{g}$)
1001	XNOR(f, g)	$\overline{f \oplus g}$	ite(f, g, $\bar{g}$)
1010	NOT(g)	$\bar{g}$	ite(g, 0, 1)
1011	f ≥ g	$f + \bar{g}$	ite(f, 1, $\bar{g}$)
1100	NOT(f)	$\bar{f}$	ite(f, 0, 1)
1101	f ≤ g	$\bar{f} + g$	ite(f, g, 1)
1110	NAND(f, g)	$\overline{fg}$	ite(f, $\bar{g}$, 1)
1111	1	1	1

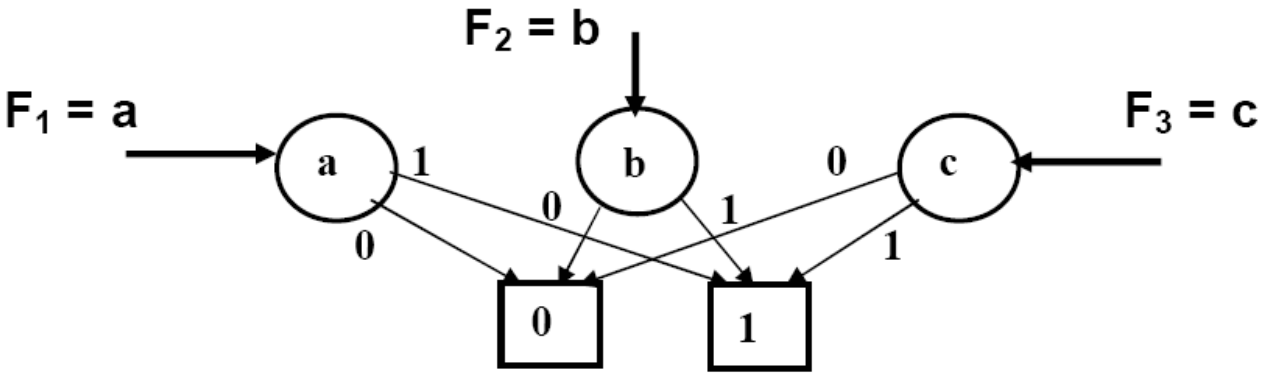
Primjer izgradnje ROBDD-a ITE-algoritmom

- Izgradi ROBDD za funkciju: ***F = ac + bc***

  uz uređenost varijabli: ***a < b < c (prva a, zadnja c)*.**

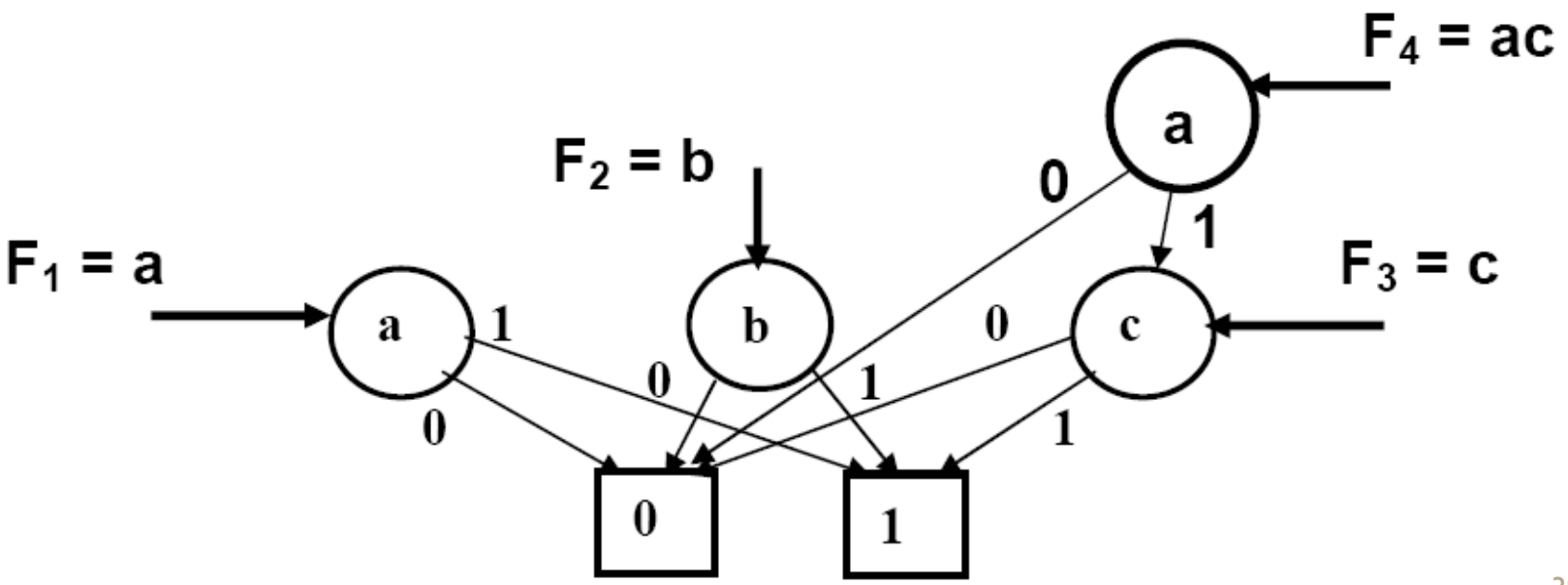
- **<u>1. korak</u>**: izgradnja elementarnih ROBDD-ova za svaku varijablu uporabom ***ite(var, 1, 0)***.

- Čvorovi ***a***, ***b***, ***c*** predstavljaju korijene ROBDD-ova projekcijskih funkcija (funkcija jedne varijable) $\mathbf{F_1 = a}$, $\mathbf{F_2 = b}$, $\mathbf{F_3 = c}$. Završni elementarni čvorovi **0** i **1** su zajednički (izomorfni grafovi).



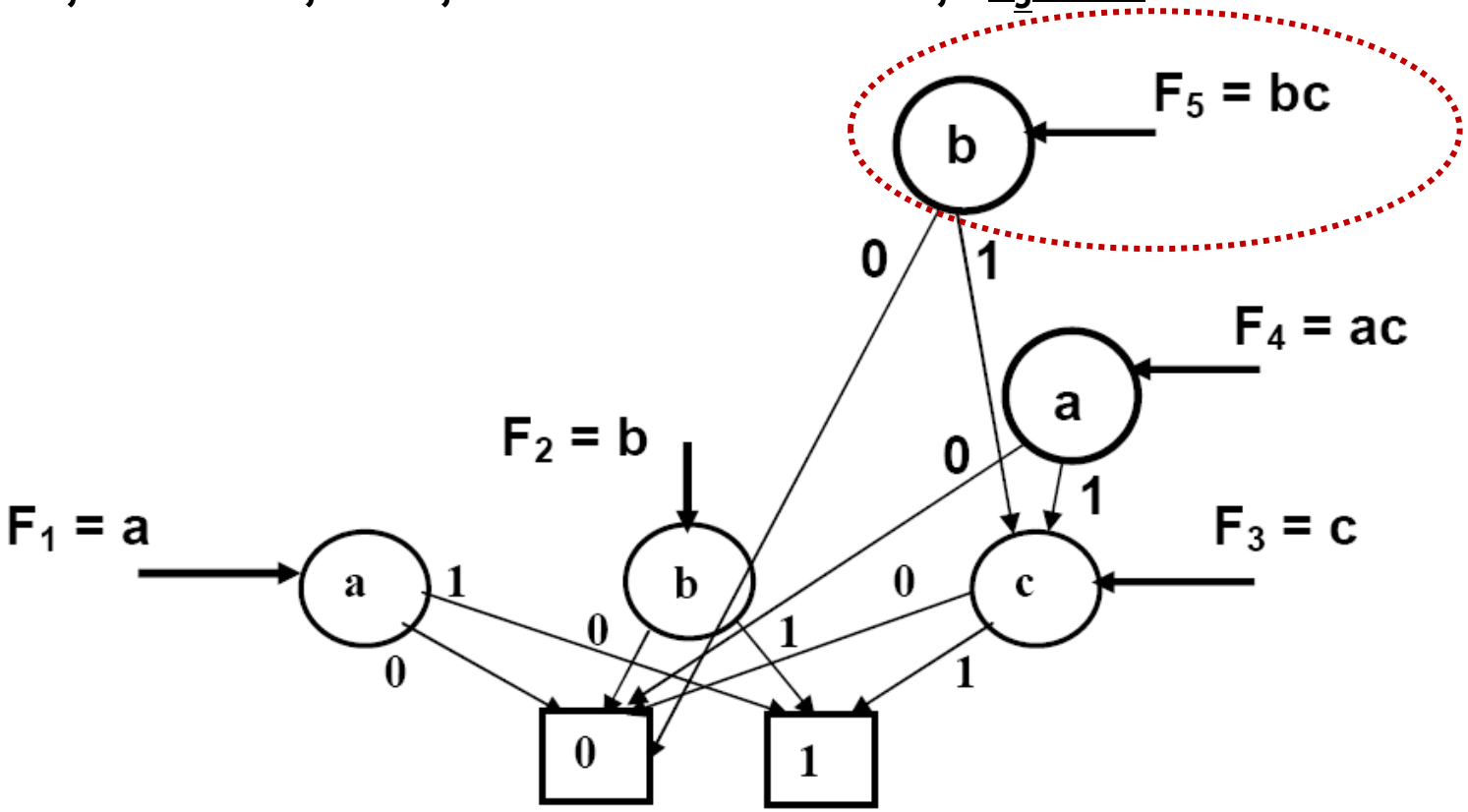
Primjer izgradnje ROBDD-a ITE-algoritmom

- **<u>2. korak</u>**: za produkt ***(ac)*** uporabom ***(ac) = a ∧ c = ite(a, c, 0)*** . Varijabla ***a*** je **samostalna**. To je dijagram s **vršnom varijablom *a***, THEN stranom ***c***, ELSE stranom ***0***. Čvorovi za THEN stranu ***c*** i ELSE stranu ***0*** već postoje. U dijagram se dodaje **novi čvor *a*** koji se povezuje na već postojeći korijenski čvor funkcije F_3**=c** i završni čvor ***0***.
- Taj novi čvor je korijen ROBDD-a za funkciju ***<u>F_4 = ac</u>***.



Primjer izgradnje ROBDD-a ITE-algoritmom

- **<u>3. korak</u>**: za produkt ***(bc)*** uporabom ***(bc) = b ∧ c = ite(b, c, 0)*** . To je dijagram s **vršnom varijablom *b***, THEN stranom ***c***, ELSE stranom ***0***. Čvorovi za THEN stranu ***c*** i ELSE stranu ***0*** već postoje. U dijagram se dodaje **novi čvor *b*** koji se povezuje na postojeći korijenski čvor F_3**=*c*** i ***0***. Taj novi čvor je korijen ROBDD-a za funkciju F_5 **= *bc***.



Primjer izgradnje ROBDD-a ITE-algoritmom

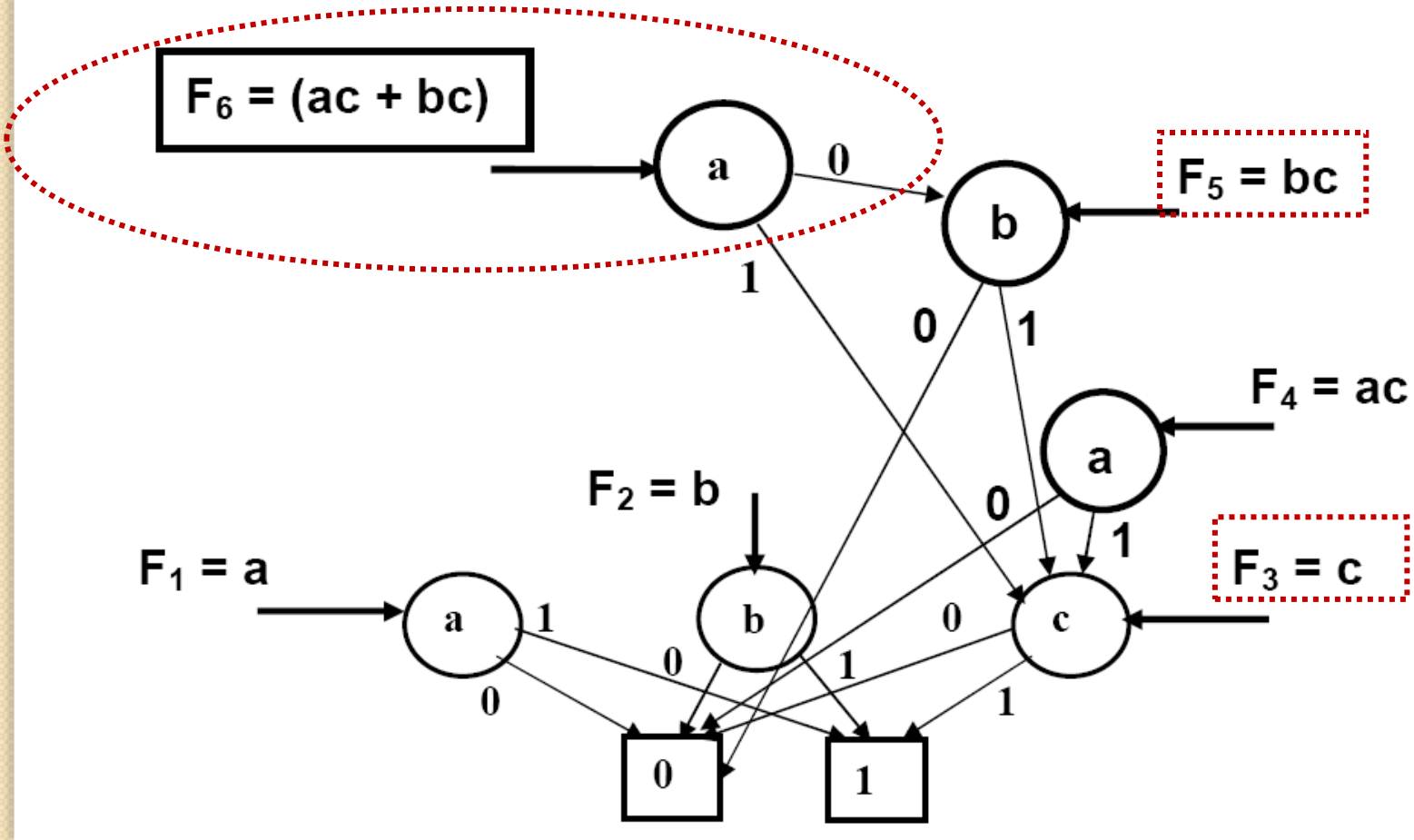
- **<u>4. korak</u>**: izgradnja ROBDD-a za cijelu funkciju $F = ac + bc$, uporabom $(ac + bc) = (ac \vee bc) = ite(ac, 1, bc)$.
- Vršna varijabla u ***ite*** funkciji **nije samostalna** pa je potrebno rastavljanje ***ite*** funkcije rekurzijom pazeći na **uređenost** varijabli.

$$ite(f, g, h) = a\ ite(f_a, g_a, h_a) + a'\ ite(f_{a'}, g_{a'}, h_{a'}) =$$
$$= ite[a,\ ite(f_a, g_a, h_a),\ ite(f_{a'}, g_{a'}, h_{a'})]$$

- U našem primjeru: $f=ac, g=1, h=bc$.
- Kofaktori tih funkcija za varijablu ***a*** su: $f_a = c,\quad g_a = 1,\quad h_a = bc$
  $f_{a'} = 0,\quad g_{a'} = 1,\quad h_{a'} = bc$
- Što rezultira u: $ite(ac, 1, bc) = ite[a, ite(c, 1, bc), ite(0, 1, bc)]$
- Neke ROBDD-ove za ***ite*** funkcije možemo pojednostaviti:
  $ite(c, 1, bc) = c$, jer to znači ako $c=1$ THEN 1, ELSE $c=0$ pa je $bc=0$
  $ite(0, 1, bc) = bc$, jer to znači ako 0, onda to nužno mora biti bc.
- Konačno: $ite(ac, 1, bc) = ite(a, c, bc)$
- To je ROBDD s korijenskim čvorom ***a***, T lukom u funkciji ***c*** i E lukom u ***bc***.

Primjer izgradnje ROBDD-a ITE-algoritmom

- **<u>5. korak</u>**: izgradnja ROBDD-a za cijelu funkciju ***F = ac + bc***, uporabom ***ite(a, c, bc)*** sastoji su u dodavanju **novoga čvora *a*** koji se povezuje na postojeće korijenske čvorove funkcija $F_3 = c$ i $F_5 = bc$. Taj čvor ***a*** je korijenski čvor ROBDD-a za zadanu funkciju $F_6 = ac + bc$.



Zaključci o postupku izgradnje ROBDD-a

- Primjer izgradnje ROBDD-a pokazao je da:

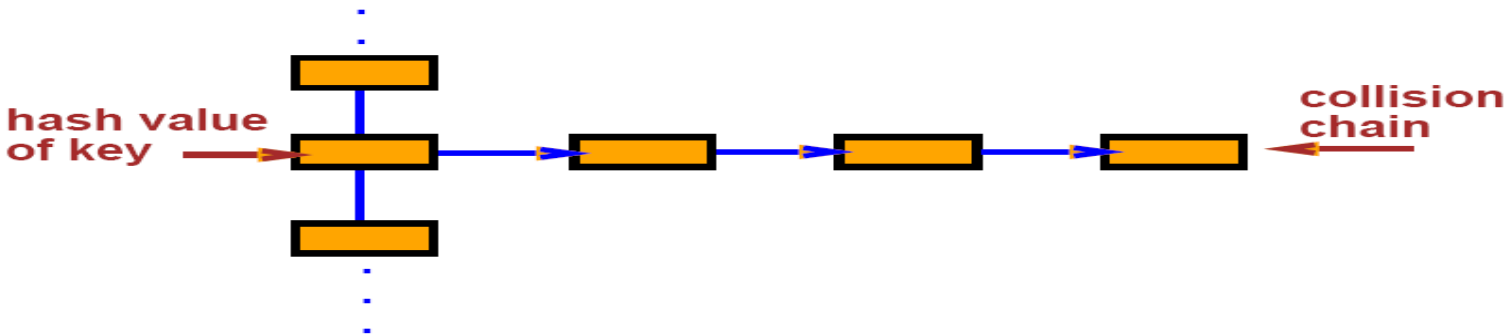
1. Svaki čvor u ROBDD-u predstavlja korijenski čvor ROBDD-a jedne jedinstvene logičke funkcije.
2. Najprije se grade jednostavni ROBDD-i za krajnje funkcije (npr. za varijable a, b, c u primjeru)
3. Tijekom izgradnje ROBDD-a za neku novu funkciju, potrebno je u svakom koraku provjeriti postoji li već možda traženi ROBDD za jednostavniju funkciju (kofaktor) s odgovarajućim korijenskim čvorom. Da bi se to omogućilo, **svaki čvor s podstablom u ROBDD-u mora biti jedinstveno označen**.

ROBDD – jedinstvena tablica

- Algoritmi izgradnje ROBDD-ova pohranjuju **sve čvorove** (***v, g, h***), gdje je ***v*** oznaka (indeks) vršne varijable u čvoru, a ***g*** i ***h*** su jedinstveni opisi (indeksi ili pokazivači) THEN i ELSE lukova iz te varijable. Potpuna struktura opisa čvora pohranjuje se u **jedinstvenoj tablici** (engl. *unique table*).
  - Ako traženi čvor već postoji u tablici, u daljnjoj izgradnji koristi se indeks ili pokazivač (engl. *pointer*) na taj čvor u jedinstvenoj tablici.
  - Ako traženi čvor ne postoji u tablici, zapisuje se taj čvor u jedinstvenu tablicu i vraća se novi indeks (pokazivač).
  - Time se ostvaruje kanonski zapis, jer za neki čvor ***f = (v, g, h)*** postoji **jedan i samo jedan** indeks (pokazivač), a to je adresa strukture opisa toga čvora u jedinstvenoj tablici.
  - Za efikasniju pohranu u memoriji računala na indekse vršnih varijabli i na indekse njihovih odgovarajućih THEN i ELSE strana primjenjuje se **funkcija "hash"**.

ROBDD – jedinstvena tablica

- Poznato je da funkcija "hash" nije savršena, pa se može dogoditi da indeksi (ključevi, engl. *key*) različitih čvorova rezultiraju u istoj adresi elementa jedinstvene tablice. Stoga se na istoj adresi jedinstvene tablice može formirati **kolizijski lanac** (lanac je implementiran povezanom listom).
- Opis svakog čvora u jedinstvenoj tablici mora se proširiti s eventualnim pokazivačem na sljedeći čvor u kolizijskom lancu.

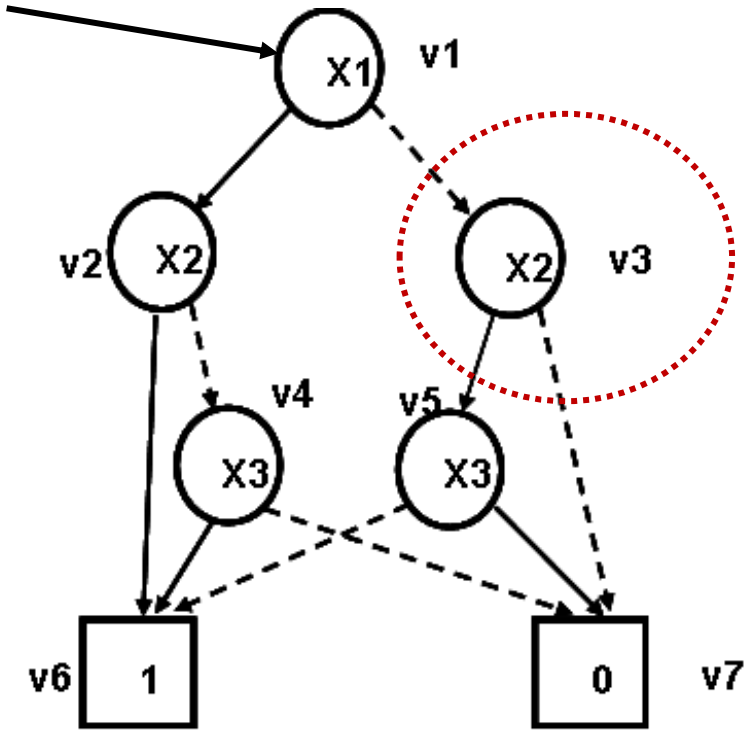


Opis svakog čvora u jedinstvenoj tablici je struktura podataka koja sadrži:

- indeks samostalne varijable
- indeks THEN strane
- indeks ELSE strane
- pokazivač na sljedeću strukturu u kolizijskom lancu

ROBDD – jedinstvena tablica - primjer

- Neka je zadana funkcija $f = (x1, x2, x3)$ predstavljena ROBDD-om.
- Svaki čvor jedinstveno je određen indeksima varijable x_i, THEN strane v_j i ELSE strane v_k, te pokazivačem na sljedeću strukturu u kolizijskom lancu.
- Npr. čvor v3 je određen: (x2, v5, v7)



- Jedinstvena tablica je niz od 5 lokacija, (od 0 do 4).
- Moguća "hash" funkcija:

$$H(x_i, v_j, v_k) = (i + j + k) \pmod 5$$

- Npr. za čvor **v3**: $H = 2+5+7 \pmod 5 = 4$

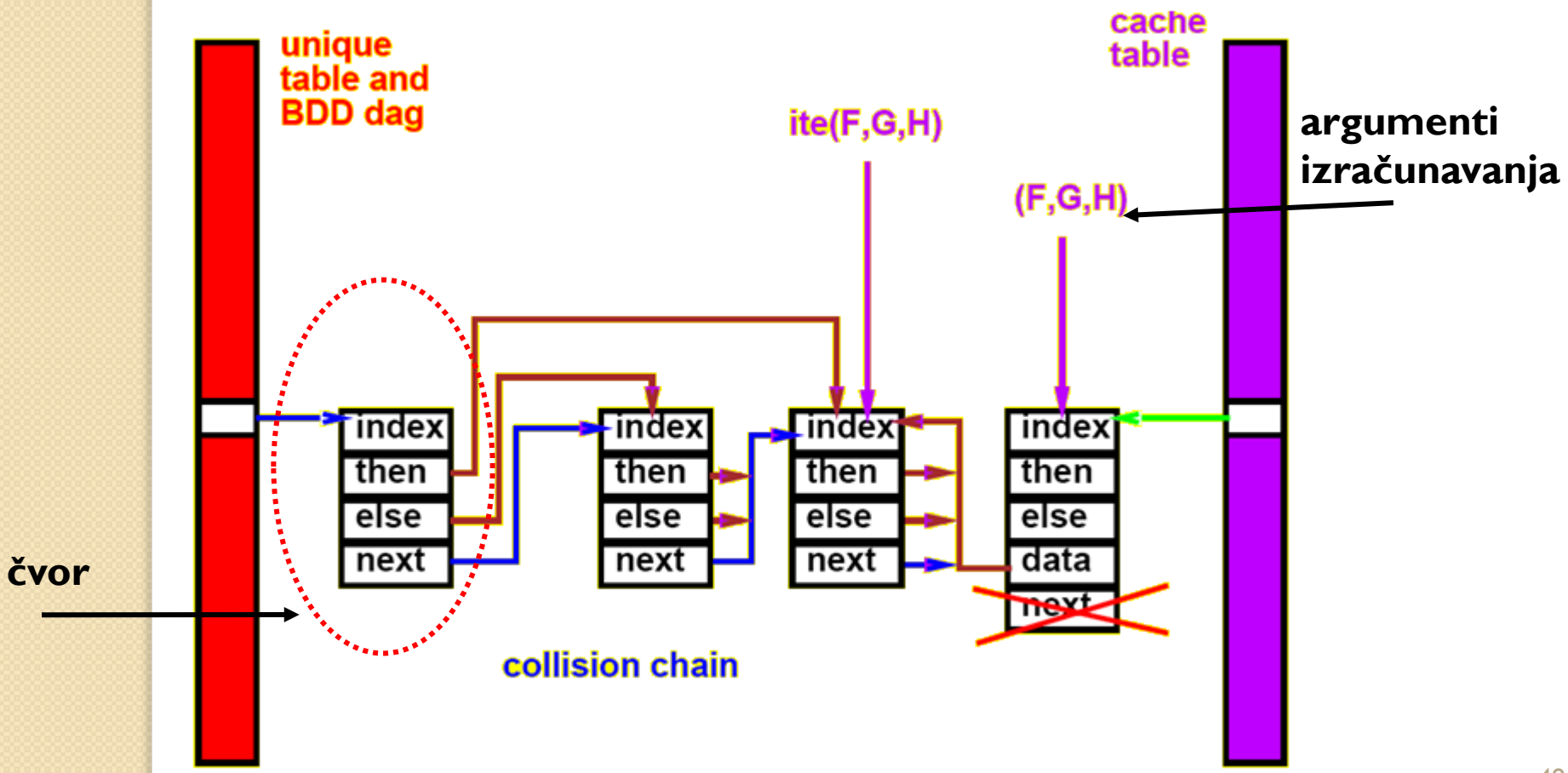
- Čvorovi **v1**, **v4**, **v5** imaju istu vrijednost funkcije "hash" (ostatak=1) pa je za njih potrebno formirati kolizijski lanac.
- Kako bi se smanjio broj kolizija, potrebno je pametno odabrati "hash" funkciju (obično da daje uniformnu razdiobu vrijednosti).

ROBDD – izračunska tablica

- Pored **jedinstvene tablice**, daljnje ubrzanje izgradnje i manipulacije s ROBDD-ovima postiže se uporabom još jedne tablice – **izračunske tablice** (engl. *computed table*), "cache", priručne tablice.
- **Jedinstvena** tablica odgovara na pitanje: "**Postoji li jedinstveni čvor** s vršnom varijablom ***v***, THEN stranom ***g*** i ELSE stranom ***h***, tj. jedinstveni čvor ***(v, g, h)*** ?"
- **Izračunska tablica** odgovara na pitanje: "**Je li nedavno izračunata neka funkcija?**", (npr. konjunkcija dviju logičkih funkcija pomoću ***ite(f, g, h)*** operatora). Time možemo izbjeći ponovno računanje ROBDD strukture koja predstavlja traženi rezultat.
- Rezultat izračunavanja je konačno **jedan ROBDD** pa izračunska tablica privremeno pohranjuje argumente izračunavanja i pokazivač na jednaku strukturu podataka kao i jedinstvena tablica, tj. na čvor ***(v, g, h)***.
- **Bitna razlika: izračunska tablica ne koristi kolizijski lanac – u slučaju kolizije stariji podatak se odbacuje.**

ROBDD – jedinstvena i izračunska tablica

- Budući da pokazuju na istu strukturu, jedinstvena i izračunska tablica mogu spojiti pokazivače na ROBDD čvorove



ITE-algoritam (uz korištenje jedinstvene i izračunske tablice)

Terminal cases:

$$(0, g, f) = (1, f, g) = f$$
$$ite(f, g, g) = g$$

$ite(f, g, h)$

```
    if(terminal case){      /* možda "terminal case", a ako nije, pogledaj je
        return result;      li to već izračunato u izračunskoj tablici */
    } else if (computed-table has entry (f, g, h){
        return result;
    }else{
        let v be the top variable of (f, g, h);
        f̃ ← ite(f_v, g_v, h_v);                    /* rekurzivno izračunavanje
        g̃ ← ite(f_v̄, g_v̄, h_v̄);                    do "terminal case" */
        if (f̃ equals g̃) return g̃;
        R ← find_or_add_unique_table(v, f̃, g̃);     // postoji li novi čvor
        insert_computed_table({f, g, h}, R);       /* dodaj argumente i
        return R; } }                              rezultat R u izračunsku
                                                   tablicu i vrati R */
```

Normalizacija ITE-trojki i ekvivalencije

- Pojednostavljenja:

$$
\begin{aligned}
ite(F,F,G) &\Longrightarrow ite(F,1,G)\\
ite(F,G,F) &\Longrightarrow ite(F,G,0)\\
ite(F,G,\overline{F}) &\Longrightarrow ite(F,G,1)\\
ite(F,\overline{F},G) &\Longrightarrow ite(F,0,G)
\end{aligned}
$$

- Ekvivalencije:

$$
\begin{aligned}
ite(F,1,G) &\equiv ite(G,1,F)\\
ite(F,0,G) &\equiv ite(\overline{G},0,\overline{F})\\
ite(F,G,0) &\equiv ite(G,F,0)\\
ite(F,G,1) &\equiv ite(\overline{G},\overline{F},1)\\
ite(F,G,\overline{G}) &\equiv ite(G,F,\overline{F})
\end{aligned}
$$

Koristiti lijevi zapis

$$ite(F,G,H) \equiv ite(\overline{F},H,G) \equiv \overline{ite(F,\overline{G},\overline{H})}, \equiv \overline{ite(\overline{F},\overline{H},G)}$$

Ekvivalencije treba odabrati tako da prvi argument ima **raniju varijablu u redoslijedu uređenosti**

Složenost izračunavanja ITE(f, g, h)

- **Bez uporabe izračunske tablice**
  - Jedan pristup jedinstvenoj tablici (bez rekurzije) = konst. vrijeme.
  - Rekurzivni poziv (ako nije završni) traži 2 nova pristupa tablici.
  - Vrijeme izvođenja je **eksponencijalno** prema broju varijabli.
- **Uz uporabu izračunske tablice i s neograničenom memorijom**:
  - Neka je $|f|$ broj čvorova u ROBDD-u za f , (analogno za g i h).
  - Za jedinstvenu kombinaciju (f, g, h) funkcija $ite(f, g, h)$ se poziva jednom.
  - Općenito $ite(f, g, h)$ se poziva $|f| \cdot |g| \cdot |h|$ puta, pa je složenost:

    $$O(|f| \cdot |g| \cdot |h|)$$

  - Za ***ite*** funkciju s dva operanda (npr. logičke funkcije AND, OR, XOR i sl.), složenost je $O(|f| \cdot |g|)$ u najgorem slučaju.
- **Uz uporabu izračunske tablice i ograničenom memorijom**
  - **U najgorem slučaju eksponencijalno** prema broju varijabli
  - **U realnim aplikacijama bliže** $O(|f| \cdot |g| \cdot |h|)$

ROBDD i ITE – efikasnost uporabe

- Za izgrađeni ROBDD neke logičke funkcije, sljedeće se operacije mogu izvesti u konstantnom vremenu (vrlo teško za oblik SOP ili POS):
- Test na tautologiju i (ne)zadovoljivost: - samo jedan završni čvor

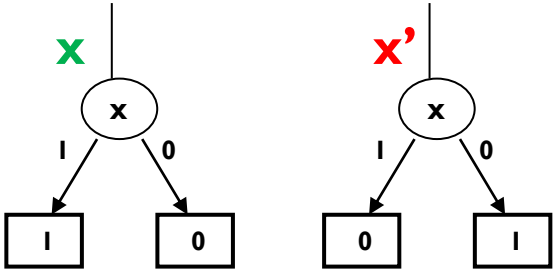
  f = 1 ? ***ite(f, 1, 0)***

  f = 0 ? ***ite(f, 0, 1)***

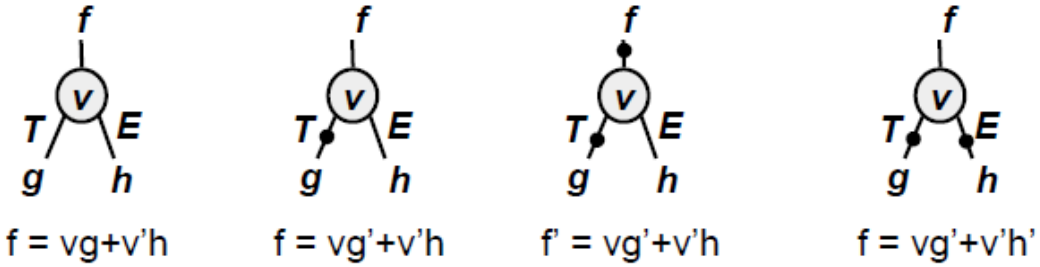
- Test na **ekvivalenciju dva ROBDD-a** provjerava za svaki čvor da su pokazivači na THEN i ELSE čvorove isti za obje funkcije – linearno ovisno samo o broju čvorova.

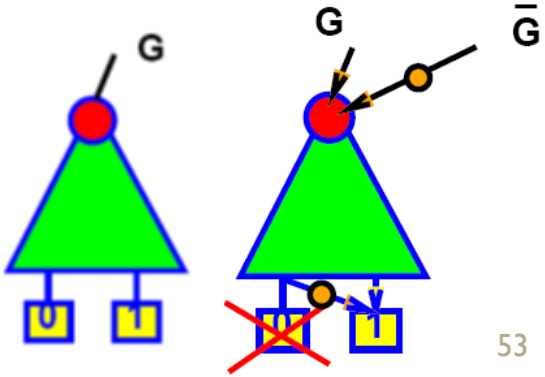
ROBDD – proširenje oznakama komplementa

- BDD-ovi za neke dvije funkcije G i G' su vrlo slični, jedina razlika su vrijednosti čvorova "djece", koji su međusobno zamijenjeni



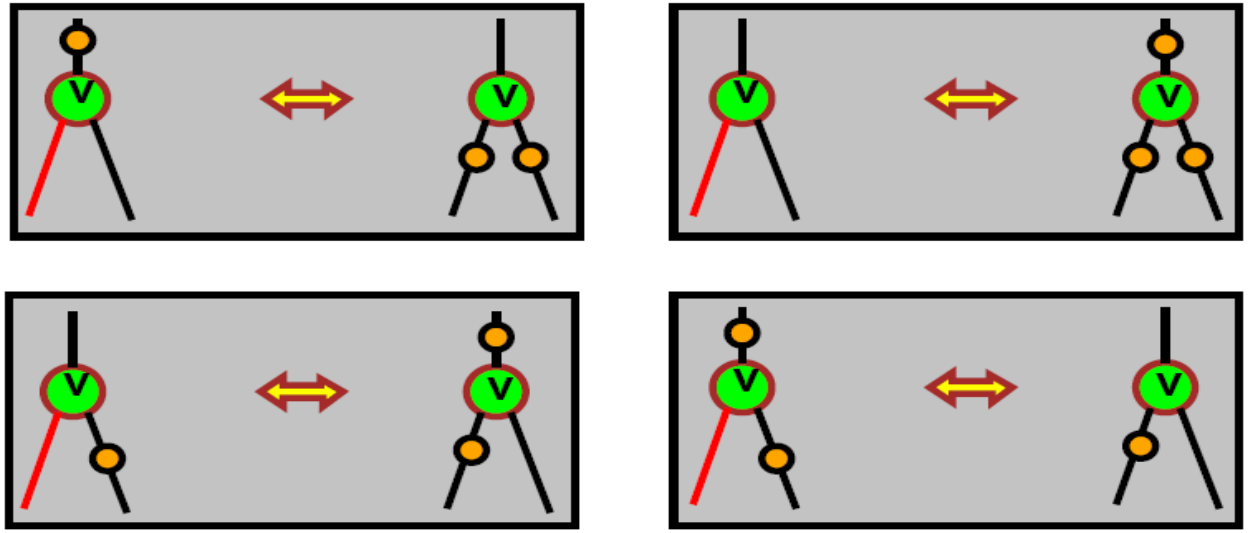
- Kako bi se iskoristio postojeći izgrađeni ROBDD za funkciju G, uvodi se komplementirani luk (pomoću punog kružića na luku) za funkciju G', čije je značenje da se **invertira rezultat** (podfunkcija) na koji pokazuje luk. To se provodi radi **optimizacije prostora i vremena pretraživanja**





ROBDD – proširenje oznakama komplementa

- Uvođenje komplementiranih lukova **narušava kanoničnost prikaza**, jer postoje logički ekvivalentni oblici funkcija. Ta nekanoničnost se mora razriješiti:



- Rješenje (konsenzus): **odaberi onaj oblik za koji komplementirani luk nije na THEN strani**

ROBDD – proširenje oznakama komplementa – postupak izgradnje

- Uvođenje komplementiranih lukova za neku funkciju *f*:
  1. Nacrta se uobičajeni ROBDD
  2. Zadrži se samo završni čvor 1, prijelazi koji idu u 0 se komplementiraju i prebace u 1
  3. Iterira se od razine zadnjih nezavršnih čvorova prema korijenu:
     1. Primijeni se ekvivalentni kanonični prikaz (vidi prethodni slajd)
     2. Ako postoje čvorovi s THEN stranom istom kao ELSE stranom, uklone se
     3. Riješe se izomorfni podgrafovi, ako postoje (ostavi se samo jedan)
  4. Kada se na taj način izgradila funkcija *f*, može se uvijek dodati u korijenu komplement za funkciju *f'*

Kao rezultat ovog postupka, ako se za neku interpretaciju varijabli prođe BDD-om kroz **paran** broj **komplementiranih** lukova, rezultat je **1**, a ako se prođe kroz **neparan** broj, rezultat je **0**.

Zadaci

3. Za Booleovu karakterističnu funkciju iz zadatka 1 nacrtajte ROBDD uz proizvoljno uređenje varijabli.

4. Za funkciju **S** sume potpunog zbrajala zadanu tablicom 1, nacrtajte ROBDD te provedite komplementiranje lukova uz uređenje x < y < cin.

Tablica 1

x	y	cin	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	0	0
0	0	1	1
0	1	1	0
1	0	1	0
1	1	1	1

5. Za funkciju **F = acd + bc + a'd'** izgradite ROBDD primjenom ITE- algoritma (rekurzivni postupak, uz potrebna pojednostavljenja) i uz uređenje a < d < c < b.
   Napomena: potrebno je raspisati cjelokupni rekurzivni postupak i nacrtati konačni ROBDD.

Zadaci

6. Odredite složenost prikaza BDD-om funkcije parnog pariteta od *n*-varijabli (dan je tablični prikaz nadesno za 4 varijable), neovisno o uređenju

x1	x2	x3	x4	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

PRIMJENA BDD-OVA

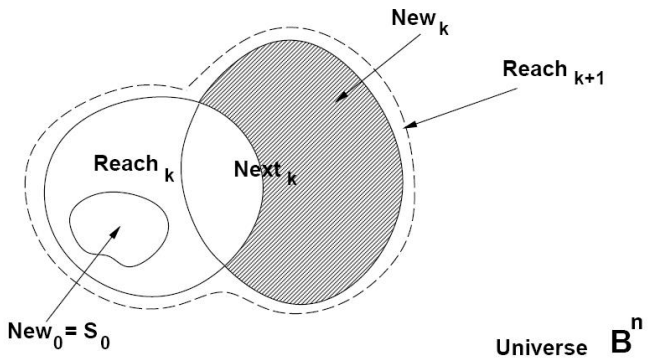
Simbolička analiza dosezljivosti BDD-ovima

- **Izračunavanje sa skupovima preslikava se u izračunavanje ROBDD-ovima**
- Skupovi ***S*** i ***T*** dani su karakterističnim funkcijama (kodiranjem) χ_S i χ_T, a one s ROBDD-ovima
- ROBDD-ovi se mogu implementirati ITE-algoritmom

Set	BDD
$\varnothing$	`BDD_0`
$\overline{S}$	`bdd_not(S)`
$S \cup T$	`bdd_or(S,T)`
$S \cap T$	`bdd_and(S,T)`
$S = T$	`S=T`
Universe	`BDD_1`

***ite* izračun**
0
ite(S, 0, 1)
ite(S, 1, T)
ite(S, T, 0)
ite(S, T, T')
1

Simbolička analiza dosezljivosti




```
2^S explore (S0,H)
{
  k:=1;  Reach:=∅;
  do {
    New_Reach:=S0 ∪ H(Reach);
    if (New_Reach = Reach)
      return Reach;
    k++;
    Reach:=New_Reach;
  } forever;
}
```

```
BDD explore (BDD S0, BDD (*H)(BDD))
{
  Reach:=BDD_0;
  do {
    New_Reach:=bdd_or(S0,H(Reach));
    if (New_Reach=Reach)
      return Reach;
    Reach:=New_Reach;
  } forever;
}
```

Simbolička provjera modela ROBDD-om

- Razmatramo provjeru modela za specifikaciju zadanom **CTL formulom i koristimo teoriju fiksne točke**

1. **Izračun formule: EX p**
   - Logička formula ***p*** dana je svojim ROBDD-om (skupom stanja u kojima ***p* = *true***).

```
BDD EX (BDD p)
{
    return H⁻¹(p);
}
```

$H(p) = \{t \in S \mid \exists s \in (p \cap R)\}$

```
BDD H(BDD p, BDD R) {
return bdd_exist(s, bdd_and(p, R))}
```

s je vektor svih stanja u zadanom početnom skupu. Disjunkcija kofaktora, isključuje početna stanja **s:**

`bdd_exist(s, f)`: $\exists_s f = f_s + f_{s'} = ite(f_s, 1, f_{s'})$

   - Kako je $H: \{s\} \rightarrow \{t\}$, to je $H^{-1}: \{t\} \rightarrow \{s\}$.
   - Operacija se u ROBDD-u postiže jednostavnom zamjenom s_i sa t_i. To je moguće napraviti jednim prolazom kroz ROBDD ako je redoslijed varijabli: $s_1, t_1, s_2, t_2, \ldots$

Simbolička provjera modela ROBDD-om

2. <u>Izračun formule CTL vremenske logike: EG p</u>

- Logička formula ***p*** dana je svojim ROBDD-om

  F (Z) = Z

- Izračun skupovima: Q(EG p) = Q(p) ∩ Q(EX EG p)
- Izračun ROBDD-ovima:

```
BDD EG (BDD p) {
    k:=1;  Z_k:=p;
      do  {
            Z_{k+1}:= bdd_and(p, H^{-1}(Z_k));
            if (Z_{k+1} = Z_k) return Z_k ;
            k++;
            } forever; }
```

Simbolička provjera modela ROBDD-om

3. <u>Izračun formule CTL vremenske logike:</u>

<u>E(p U q) = (p EU q)</u>

- Logičke formule ***p*** i ***q*** dane su svojim ROBDD-ovima.

F (Z) = Z

- Izračun skupovima: $Q(p\ EU\ q)) = Q(q) \cup Q(p) \cap Q(EX\ (p\ EU\ q))$
- Izračun ROBDD-ovima:

```
BDD EU (BDD p, BDD q) {
    k:=1;  Z_k:=q;
      do  {
            Z_{k+1}:= bdd_or(q, bdd_and(p, H^{-1}(Z_k)));
            if (Z_{k+1} = Z_k) return Z_k ;
            k++;
            } forever; }
```

Prisjetimo se ekvivalencija...

- $AX\ \varphi \quad = \quad \neg EX\ \neg\varphi$
- $AF\ \varphi \quad = \quad \neg EG\ \neg\varphi$
- $AG\ \varphi \quad = \quad \neg EF\ \neg\varphi = \neg E\ (True\ U\ \neg\varphi)$

Dodatno:

- $A\ (\psi\ U\ \varphi) = \neg E\ (\neg\varphi\ U\ \neg\psi \wedge \neg\varphi) \wedge AF\ \varphi$

- Stoga je moguće sve ostale vremenske formule izračunati uz pomoć postojeća tri algoritma EX, EG, EU koja računaju s BDD-ovima

Kako to ide s BDD-ovima u NuSMV-u

1. Pokrene se NuSMV u interaktivnom načinu
   > NuSMV -int
2. Učita se datoteka s modelom
   > read_model –i file.smv
3. Sravni se hijerarhija modula, čime se instanciraju moduli i procesi
   > flatten_hierarchy
4. **Varijable modula i procesa se kodiraju**, čime se dobivaju njihove karakteristične funkcije
   > encode_variables (po defaultu, stvaraju se prolaskom po dubini modela)
5. Za kodirane varijable gradi se ukupni BDD slijedeći odgovarajuće relacije prijelaza i početna stanja
   > build_model
6. Uz pomoć izgrađenog BDD-a provjerava se dosezljivost stanja, provjera specifikacije i drugo
   > compute_reachable , check_fsm , check_ctlspec...

Redoslijed varijabli u NuSMV-u

- Po *defaultu*, prilikom poziva naredbe `encode_variables` redoslijed varijabli je određen prolaskom u dubinu kroz hijerarhiju modela
- Može se navesti ulazna datoteka s eksplicitno zadanim poretkom varijabli `encode_variables -i input_order_file`
- Moguće je uvesti heuristiku *basic* kojom se grupiraju varijable koje su zajedno u prijelazima Kripke strukture (`bdd_static_order_heuristics`)
- Dinamičko preuređivanje događa se ako broj čvorova dostigne neki prag ili ako korisnik to eksplicitno pokrene
- Naredba za pokretanje dinamičkog preuređivanja je `dynamic_var_ordering -e <method>`, gdje je `<method>` neka od metoda iz skupa:
  - *Sift* – pomiče jednu po jednu varijablu kroz poredak dok ne nađe najbolji poredak
  - *Random* – parovi slučajno odabranih varijabli zamijene mjesta
  - *Window* – permutira varijable unutar prozora od 2,3 ili 4 susjedne varijable
  - Ostalo (*genetic*, *annealing*...)

Zaključak

- ROBDD-ovi su zanimljive i uporabive strukture podataka.
- Zahvaljujući ROBDD strukturama, mnogi verifikacijski problemi mogu se uspješno riješiti.
- *"There is no such thing as a free lunch"*: Postoje arhitekture sklopova (npr. sklop za množenje), te pojedinih programskih dijelova koje se ni sa ROBDD-ovima ne mogu riješiti u polinomnom vremenu i prostoru
- Najpoznatiji programski paketi:
  - **CMU BDDLIB**: Carnegie Mellon University, Pittsburgh.
  - **CUDD**: BDD package, University of Colorado, Boulder.
- ROBDD-ovi su uključeni u većinu suvremenih alata za provjeru modela i provjeru ekvivalencije

Zadatak za bonus bod

- Nacrtajte ROBDD dijagram pojednostavljen s komplementiranim lukovima za logičku funkciju:

$$F(x1,x2,x3) = (x1 + x2)(x1 + x3)(x2 + x3)(x1' + x2')$$

uz uređenost varijabli $x3 < x1 < x2$.

FVPP: SAT-rješavači

Algoritmi DPLL, GRASP, Chaff i MiniSAT
Primjena u provjeri modela.

Pripremio: izv. prof. dr. sc. Alan Jović

Ak. god. 2022./2023.

Sadržaj

- SAT-problem
- Temeljni algoritam DPLL
- Algoritam GRASP
- Algoritmi Chaff i MiniSAT
- Primjena SAT-rješavača u FV

SAT-PROBLEM

SAT-problem

- Temeljni NP-kompletan problem
- Traži se **model** (interpretacija koja se evaluira u istinu) skupa formula
- Skup formula zadan najčešće u **CNF-obliku** (konjunkcija više klauzula):

$$\Gamma = K_1 \wedge K_2 \wedge \ldots \wedge K_m$$
$$= (k_{11} \vee \ldots \vee k_{1p}) \wedge (k_{21} \vee \ldots \vee k_{2r}) \wedge \ldots \wedge (k_{m1} \vee \ldots \vee k_{ms})$$

- K_i – *i*-ta klauzula; k_{ij} – *j*-ti literal u *i*-toj klauzuli
- Složenost SAT-problema za 3CNF ili više je eksponencijalna($O(2^n)$) u ovisnosti o broju atomičkih simbola (varijabli)

SAT-problem

- Iscrpna procedura rješavanja **CNF** SAT-problema:
  - Sustavno se pridjeljuju istinitosne vrijednosti atomičkim propozicijskim simbolima sve dok se ne nađe model ili iscrpe sve mogućnosti.
  - Grananje na svakom atomičkom simbolu – istina ili laž
  - Ako se ne nađe model (pronađe se konflikt), vraća se na prethodnu varijablu i isprobava se drugu mogućnost
  - **Za *n* atoma: 2^n pridruživanja**.
- **Najgori slučaj: $O(2^n)$**.
- **Prosječni slučaj (suvremeni rješavači): $O(n^p)$.**
- Primjeri heurističkih SAT-rješavača koji garantiraju završetak pretrage:
  - DPLL, Grasp, Chaff (zChaff), MiniSAT, SatZ...

Zašto se bavimo SAT-rješavačima?

- Problemi u mnogim područjima mogu se svesti na SAT-problem
  - **Formalna verifikacija sklopovlja i programa ← fokus**
  - Optimizacijski problemi
  - Umjetna inteligencija (npr. dijagnoza kvarova), automatsko planiranje (*satplan*)
- Drugi suvremeni rješavači (npr. QBF, SMT) posuđuju tehnike korištene u SAT-rješavačima
- SAT-rješavači su još uvijek aktivno područje istraživanja, natjecanja se kontinuirano održavaju:

  http://www.satcompetition.org/

TEMELJNI ALGORITAM DPLL

DPLL

- Davis-Putnam-Logemann-Loveland, 1962.
- Predstavlja osnovu za sve suvremene SAT-rješavače
- U svakom koraku algoritma koriste se dvije heuristike:
  - **Propagacija jedinične klauzule** (engl. *Unit clause propagation*, dalje: PJK)
  - **Uklanjanje klauzula s čistim literalima** (engl. *Pure literal elimination*, dalje: UKČL)

Propagacija jedinične klauzule

- Ako je neka klauzula jedinična, **što znači da sadrži samo jedan literal,** tada se varijabli može pridružiti onu vrijednost koja čini tu klauzulu istinitom
- Takva jedinična klauzula se eliminira iz daljnjeg razmatranja, a sve one klauzule koje sadrže tu varijablu se razrješavaju
- PJK se ponavlja sve dok više nema jediničnih klauzula
- Smanjivanje broja klauzula se može u pojedinim specifičnim slučajevima svesti na kaskadu kojom se može vrlo brzo doći do modela

Propagacija jedinične klauzule

- Primjer:

$\Gamma = ((x1 \vee x2) \wedge (\neg x1 \vee \neg x2) \wedge (x1))$

$= ((T \vee x2) \wedge (F \vee \neg x2) \wedge (T))$

$= ((T) \wedge (\neg x2) \wedge (T))$

$= (\neg x2)$

$= T$

Jedinična klauzula: $x1$, zatim $\neg x2$

Rješenje: $\{x1=T, x2=F\}$

Uklanjanje klauzula s čistim literalima

- **Čisti literal** je varijabla koja se pojavljuje u samo **jednoj polarnosti** (negirana ili nenegirana) u **svim** klauzulama gdje je prisutna
- **Čistom literalu uvijek se može pridijeliti takvu vrijednost da klauzula postane istinita i da time takva klauzula prestane biti relevantna u potrazi za modelom**
- Sve takve klauzule se uklanjaju i čisti literali se pamte, što se ponavlja dokle je to moguće
- Primjer:

$\Gamma = ((x1 \vee x2 \vee x3) \wedge (\neg x1 \vee \neg x2 \vee x3) \wedge (\neg x1 \vee \neg x2) \wedge (x1 \vee \neg x2))$

x3 je čisti literal, klauzule koje ga sadrže zanemaruju se i pamti se (x3 = T). Ostaje:

$= ((\neg x1 \vee \neg x2) \wedge (x1 \vee \neg x2))$

$\neg$x2 je čisti literal, zanemaruju se obje klauzule i pamti se (x2 = F).

= T

Moguće rješenje: {x1=T, x2 =F, x3=T} - x1 može biti i F, svejedno je

DPLL

- Algoritam započinje s formulom u CNF obliku
- Algoritam u svakom koraku koristi najprije PJK dokle je moguće, a zatim UKČL dokle je moguće
- Ako nije pronađeno rješenje, treba se zatim odabrati varijablu za grananje
- Učinkovitost DPLL-a značajno ovisi o **strategiji izbora varijable za grananje**, osnovna strategija je **slučajni** izbor varijable i slučajni izbor njene vrijednosti (T ili F)
- Algoritam završava ako:
  - postoji konzistentan skup literala (našao model) ili
  - ispitane su sve mogućnosti i svugdje se naišlo na konflikt (nekonzistentan skup literala)

Osnovna strategija pretraživanja s povratkom na prethodnu varijablu

1. Donesi odluku o varijabli grananja i pridruži joj vrijednost
2. Zaključi implicirana pridruživanja deduktivnim procesom (PJK i UKČL).
   - Može dovesti do nekonzistentnih klauzula: **konflikt!**
   - Pridruživanje koje je dovelo do konflikta zove se konfliktno pridruživanje.
3. Ako dođe do konflikta, pokušava se pridruživanje alternativne vrijednosti varijabli grananja (kronološki povrat, tzv. *backtracking*)
4. Ako opet dođe do konflikta, vraća se slijedno nazad uz hijerarhiju grananja i provode se pridruživanja alternativne vrijednosti varijabli grananja
5. Ako su isprobane sve mogućnosti varijabli grananja i svugdje je došlo do konflikta, onda problem nije zadovoljiv

Poboljšanja DPLL-a

- Poboljšanja DPLL-a išla su u smjeru:
  - Uvođenja raznih heuristika pri izboru literala za grananje
  - Variranja osnovne strategije povratka na prethodnu varijablu pri konfliktu
  - Optimiranja propagacije ograničenja u klauzulama
  - Optimiranja struktura podataka koje se koriste za implementaciju

ALGORITAM GRASP

GRASP

- **GRASP** - Generalized seaRch Algorithm for the Satisfiability Problem (Silva, Sakallah, 1996.)
- Koristi se heuristika za varijablu grananja: pridijeli svakoj varijabli obje mogućnosti, a zatim odredi koje će pridruživanje zadovoljiti najviše klauzula (2*n* pridjeljivanja * provjera *p* klauzula)
  - Time se ustvari pronalazi najčešći literal
- Značajke algoritma:
  - **Učenje novih klauzula kroz konflikte** (engl. *conflict-driven clause learning*, CDCL).
  - **Grafovi implikacija** za propagaciju jediničnih klauzula i analizu konflikata.
  - Analiza konflikata i grafovi implikacije dovode do **ne-kronološkog** povrata na prethodnu varijablu (tzv. *backjumping*)!

Grafovi implikacija

- PJK je specijalan slučaj propagacije Booleovih ograničenja (engl. *Boolean Constraint Propagation*, PBO)
- GRASP radi PBO koristeći **grafove implikacija** (engl. *implication graph*)
  - npr. $K_i = (x \vee \neg y)$, ako je pri grananju $y = 1$ tada je $x = 1$ **impliciran** ili **forsiran**, zato što je nužno da bude $x = 1$ da bi formula bila SAT
- PBO je iterativna primjena implikacija sve dok to više nije moguće ili je došlo do konflikta

Grafovi implikacija

- **Pridruživanje prethodnika** varijable x, u oznaci A(*x*) je pridruživanje vrijednosti 0 svim drugim literalima osim x u toj klauzuli, čime je x forsiran na vrijednost 1.
- Primjer
  - $K_j = (x \vee y \vee \neg z)$,

    $A(x) = \{y{:}0, z{:}1\}$, $A(y) = \{x{:}0, z{:}1\}$, $A(z) = \{x{:}0, y{:}0\}$
  - Pridruživanje prethodnika direktno "prokazuje" vrijednosti varijabli koje su prouzročile forsiranje varijable "x" na 1.

Grafovi implikacija

- Grafovi implikacija prikazuju **odluke o grananju** i **implicirana pridruživanja**.
- Prethodnici čvora *x* u grafu implikacija odgovaraju pridruživanju prethodnika A(*x*), a povezani su s čvorom *x* lukovima na kojima je označena jedinična klauzula (implikacija) koja je dovela do *x*.
  - U grafu implikacija ne prikazuju se prethodnici čvora ako je čvor *x* odluka o grananju!
- Za specijalan konfliktni čvor κ (kappa), pridruživanje prethodnika $A(\kappa)$ je pridruživanje varijabli u klauzuli koja je nekonzistentna i time neistinita.

Primjer grafa implikacija

Primjer prilagođen od: W. Klieber, CMU, 2011.

Oznaka razine odluke. Neka pridruživanja bile su grananja, a neke implikacije (forsiranja)

$K_1 = (\neg x_1 \vee x_2)$

$K_2 = (\neg x_1 \vee x_3 \vee x_9)$

$K_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$

$K_4 = (\neg x_4 \vee x_5 \vee x_{10})$

$K_5 = (\neg x_4 \vee x_6 \vee x_{11})$

$K_6 = (\neg x_5 \vee \neg x_6)$

$K_7 = (x_1 \vee x_7 \vee \neg x_{12})$

$K_8 = (x_1 \vee x_8)$

$K_9 = (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$

...

- Trenutačno pridruživanje istinitosti:

  $\{x_9=0@1, x_{12}=1@2, x_{13}=1@2, x_{10}=0@3, x_{11}=0@3, \ldots\}$

- Trenutačna odluka pri pridruživanju: $\{x_1=1@6\}$

- **Ideja:** pronaći klauzulu koja, uz trenutačnu odluku pri pridruživanju i ostalih trenutnih pridruživanja istinitosti, može dovesti do impliciranja (forsiranja) određenog literala u toj klauzuli. Nakon toga, nacrtati pridruživanja kao čvorove koji su doveli do impliciranja literala i povezati ih s čvorom impliciranog literala strelicom s oznakom klauzule. Postupak nastaviti uzimajući u obzir implicirane literale dok se ne dođe do konflikta, do rješenja problema ili do potrebe za grananjem.

Napomena: i klauzula i varijabli u trenutačnom pridruživanju ima još, ali nam nisu bitni za rješenje ovog primjera (primijetiti tri točke ...)

Primjer grafa implikacija

Primjer prilagođen od: W. Klieber, CMU, 2011.

Oznaka razine odluke. Neka pridruživanja bile su grananja, a neke implikacije (forsiranja)

- Trenutačno pridruživanje istinitosti:

  $\{x_9=0@1,\ x_{12}=1@2,\ x_{13}=1@2,\ x_{10}=0@3,\ x_{11}=0@3\ ,\ \ldots\}$

- Trenutačna odluka pri pridruživanju: $\{x_1=1@6\}$

$K_1 = (\neg x_1 \vee x_2)$

$K_2 = (\neg x_1 \vee x_3 \vee x_9)$

$K_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$

$K_4 = (\neg x_4 \vee x_5 \vee x_{10})$

$K_5 = (\neg x_4 \vee x_6 \vee x_{11})$

$K_6 = (\neg x_5 \vee \neg x_6)$

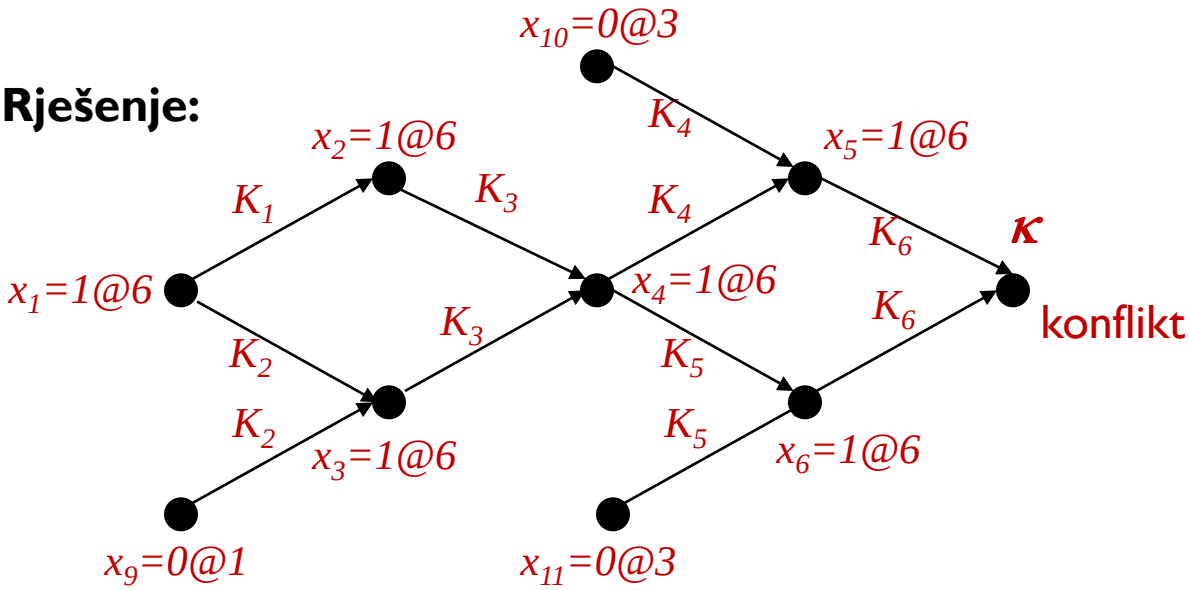
$K_7 = (x_1 \vee x_7 \vee \neg x_{12})$

$K_8 = (x_1 \vee x_8)$

$K_9 = (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$

...

Rješenje:



Napomena: i klauzula i varijabli u trenutačnom pridruživanju ima još, ali nam nisu bitni za rješenje ovog primjera (primijetiti tri točke ...)

Analiza konflikta

- Ako se dogodi konflikt, analizira se graf implikacija:
  - Dodaje se nova klauzula koja će spriječiti istraživanje istog konflikta u budućnosti
    $\Rightarrow$ **Učenje novih klauzula kroz konflikte (CDCL):**
    - Neka je KP konfliktno pridruživanje klauzule, u našem primjeru:.

      KP = $\{x_1=1@6,\ x_9=0@1,\ x_{10}=0@3,\ x_{11}=0@3\}$

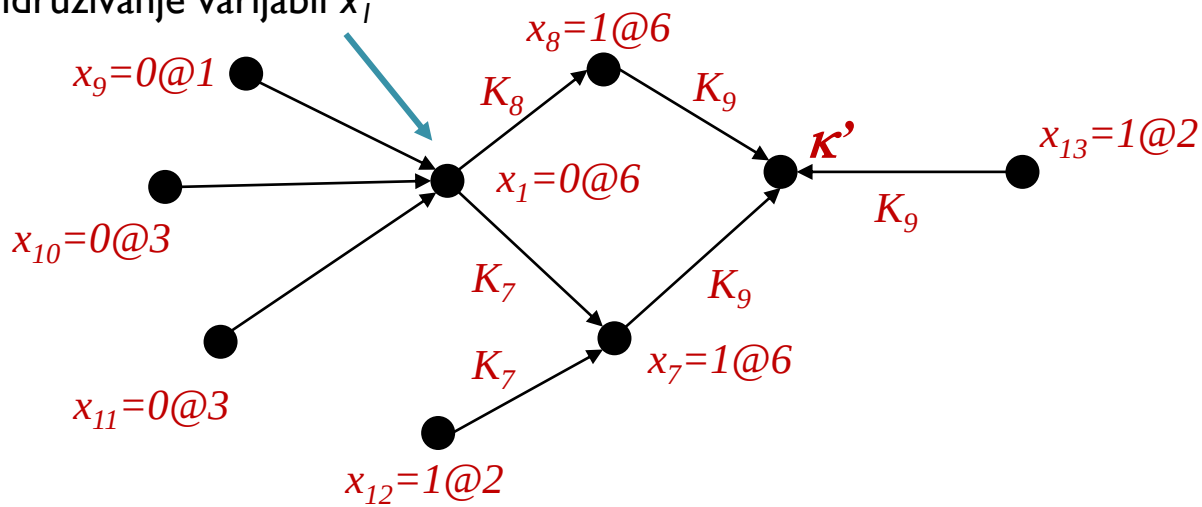
      Naučena nova klauzula koja će izbjeći isti konflikt u budućnosti jednaka je **negaciji konfliktnog pridruživanja**:

      $K_{new} = \neg KP = \neg(x_1 \wedge \neg x_9 \wedge \neg x_{10} \wedge \neg x_{11}) = \neg x_1 \vee x_9 \vee x_{10} \vee x_{11}$
  - Odredi se razina odluke do koje se treba vratiti (na koju treba skočiti), to ne mora nužno biti prethodna razina
    $\Rightarrow$ **Ne-kronološki povratak**

Analiza konflikta

- Literal $l \in$ KP je jedinstvena implikacijska točka (JIT) akko svaki drugi literal $m \in$ KP , $m \neq l$ ima raniju razinu odluke od l.
- Do JIT se dolazi uklanjanjem svih implikanata u implikacijskom grafu (konfliktnom pridruživanju KP) na zadnjoj razini odluke (u našem primjeru uklone se x_5, x_4,...) sve dok se ne dođe do zadnjeg ($x_1 = 1@6$.) U našem slučaju, literal $x_1=1$ je JIT
- Ako bismo sada zbog nastalog konflikta pridružili $x_1 = 0$, izgradnjom novog grafa implikacija pokazuje se da bi se ponovno dobio konflikt, κ', a ovaj put KP' = $\{x_9=0@1, x_{10}=0@3, x_{11}=0@3, x_{12}=1@2, x_{13}=1@2\}$:

x_9, x_{10} i x_{11} su uzrokovali alternativno pridruživanje varijabli x_1



Ne-kronološki povratak

- Ideja ne-kronološkog povratka (*backjumping*) je vratiti se na **zadnju (najdublju) razinu odluke** koja postoji kod literala unutar **KP'** (dakle, nakon što smo isprobali alternativnu vrijednost varijable grananja).
- Kod kronološkog povratka ta razina bi bila razina od JIT – 1, budući da smo isprobali obje vrijednosti varijable na razini JIT. S druge strane, *backjumping* nas u ovom slučaju dovodi na razinu odluke 3

- Na taj način drastično se smanjuje pregledavanje prostora stanja na razinama 4 i 5, jer bi i na tim razinama uvijek dolazilo do konflikta, budući da KP' ne ovisi izravno o pridruživanjima na tim razinama (klauzula izvedena i naučena iz KP': $\neg$KP' = $x_9 \vee x_{10} \vee x_{11} \vee \neg x_{12} \vee \neg x_{13}$ bi bila uvijek nezadovoljena na razinama 4 i 5)

Problemi GRASP-a

- Rast broja klauzula s brojem ne-kronoloških povrata (u općem slučaju eksponencijalno)
  - Rješenje: ograničava se veličina novo naučene klauzule na *k* literala
- Veliki fiksni trošak (engl. *overhead*) prilikom analize konflikta i prilikom odluke o varijabli grananja (svaki put treba tražiti najčešći literal)
- Heuristika koju GRASP koristi pri odabiru varijable grananja ne daje uvijek najbolje rezultate

Okvirni pseudokod GRASP-a

```
Neka je Γ formula u CNF-obliku
TrenutnoPridr = {};
while (true) {
    while (vrijednost od Γ za TrenutnoPridr nepoznata){
        OdlučiGrananje (); // Dodaj literal u TrenutnoPridr
        PBO(); // Dodaj implicirane literale u TrenutnoPridr
    }
    if (TrenutnoPridr zadovoljava Γ) {return true;}
    AnalizaKonfliktaIUčenjeNoveKlauzule();
    if (naučena klauzula je prazna) {return false;}
    PovratakNaPrethodnuVarijablu();
    PBO();
}
```

ALGORITMI CHAFF I MINISAT

Chaff

- Moskewicz et al., 2001. (suradnja UC Berkeley, MIT i Princeton University)
- Još uvijek vrhunski SAT-rješavač, poboljšavan do 2007., ugrađen u NuSMV i NuXmv (zChaff)
- Koristi neke tehnike od prethodnika (DPLL, GRASP)
- Fokus na boljem inženjerstvu svih aspekata pretrage
- Naglasak na **optimizaciji propagacije Booleovih ograničenja** (jer tu algoritmi potroše najviše, oko 80% vremena, samo oko 10% na analizi konflikta) i smanjenju fiksnog troška pri odluci o varijabli grananja

Chaff

- Značajke Chaffa
  - Učinkovita PBO
    - **Promatraju se dva literala**
    - Brzi povratak na prethodnu razinu
  - Učinkovita procedura odlučivanja o grananju
    - Lokalizira prostor pretraživanja
  - CDCL kao kod GRASP-a
  - Uklanjanje naučenih klauzula
  - Pretraga iznova (*restart*)

Učinkovita PBO

- Što određuje forsiranje vrijednosti (implikaciju) literala?
  - Svi literali u klauzuli osim jednoga iznose 0
  - Npr. za (v1 ∨ v2 ∨ v3) implikacije su: (0 ∨ 0 ∨ v3) ili (0 ∨ v2 ∨ 0) ili (v1 ∨ 0 ∨ 0)
  - Za klauzulu sa p literala, forsiranje literala se događa nakon što $p - 1$ literala imaju pridruženu vrijednost *false*
  - Ideja: mogli bismo potpuno ignorirati analiziranje prvih $p - 2$ pridruživanja toj klauzuli i reagirati tek kad se prijeđe s $(p - 2)$. na $(p - 1)$. pridruživanje
  - Praktično: za svaku klauzulu odaberu se **dva literala** na slučajan način koji će se "promatrati", a ignoriraju se pridruživanja svim ostalim literalima u toj klauzuli

Primjer

- U klauzuli (v1 ∨ v2 ∨ v3 ∨ v4 ∨ v5) promatramo v1 i v2:
- (**v1=X** ∨ **v2=X** ∨ v3 = ? ∨ v4=? ∨ v5=?), pridruživanja za v3,v4,v5 ne razmatramo u toj klauzuli

Primjer učinkovite PBO

Primjer prilagođen od T. Heyman i W. Klieber, CMU

$K1 = \underline{v2} \vee \underline{v3} \vee v1 \vee v4$

$K2 = \underline{v1} \vee \underline{v2} \vee \neg v3$

$K3 = \underline{v1} \vee \neg \underline{v2}$

$K4 = \neg \underline{v1} \vee \underline{v4}$

$K5 = \neg v1$

Početni slučajno odabrani <u>promatrani literali</u>

Primjer učinkovite PBO

Trenutačni stog pridruživanja = {v1 = F} (zbog implikacije na jediničnoj klauzuli K5)

$K1 = \underline{v2} \vee \underline{v3} \vee \mathbf{v1} \vee v4$

$K2 = \underline{\mathbf{v1}} \vee \underline{v2} \vee \neg v3$

$K3 = \underline{\mathbf{v1}} \vee \neg \underline{v2}$

$K4 = \neg \underline{\mathbf{v1}} \vee \underline{v4}$

1. Razmatraju se sve klauzule gdje odabrani literal daje vrijednost F i time smanjuje veličinu klauzule
2. Ne razmatraju se one klauzule gdje bi odabrani literal dao vrijednost T, jer je time cijela klauzula T
3. Ne razmatraju se one klauzule u kojima se odabrani literal ne promatra

Primjer učinkovite PBO

Trenutačni stog pridruživanja = {v1 = F}

$$K1 = \underline{v2} \vee \underline{v3} \vee v1 \vee v4$$

$$K2 = v1 \vee \underline{v2} \vee \underline{\neg v3}$$

$$K3 = v1 \vee \neg \underline{v2}$$

$$K4 = \neg \underline{v1} \vee \underline{v4}$$

Razmatramo 2. i 3. klauzulu.

U 2. klauzuli odabiremo $\underline{\neg v3}$ kao sljedeći promatrani literal
3. klauzula postala je ovim pridruživanjem jedinična klauzula. Forsirani literal (implikaciju) v2 = F dodajemo u red pridruživanja koja moramo obraditi

Primjer učinkovite PBO

Trenutačni stog pridruživanja = {v1 = F, v2 = F}

$$K1 = v2 \vee \underline{v3} \vee v1 \vee \underline{v4}$$

$$K2 = v1 \vee v2 \vee \underline{\neg v3}$$

$$K3 = v1 \vee \neg \underline{v2}$$

$$K4 = \neg \underline{v1} \vee \underline{v4}$$

Razmatramo 1. i 2. klauzulu, gdje odabrani literal v2 = F smanjuje veličinu klauzule.

U 1. klauzuli odabiremo v4 kao sljedeći promatrani literal
2. klauzula postala je jedinična klauzula. Implikaciju v3 = F dodajemo u red pridruživanja koja moramo obraditi

Primjer učinkovite PBO

Trenutačni stog pridruživanja = {v1 = F, v2 = F, v3 = F}

$$K1 = v2 \vee v3 \vee v1 \vee \underline{v4}$$

$$K2 = v1 \vee v2 \vee \underline{\neg v3}$$

$$K3 = v1 \vee \neg \underline{v2}$$

$$K4 = \neg \underline{v1} \vee \underline{v4}$$

1. klauzula se razmatra, gdje odabrani literal v3 = F smanjuje veličinu klauzule.

1. klauzula postala je jedinična klauzula. Implikaciju v4 = T dodajemo u red pridruživanja koja moramo obraditi.

Primjer učinkovite PBO

Trenutačni stog pridruživanja = {v1 = F, v2 = F, v3 = F, v4 = T}

$K1 = v2 \vee v3 \vee v1 \vee \underline{v4}$

$K2 = v1 \vee v2 \vee \underline{\neg v3}$

$K3 = v1 \vee \neg \underline{v2}$

$K4 = \neg \underline{v1} \vee \underline{v4}$

Budući da nema konflikta i nemamo više literala u redu obrade, PBO završava i treba se donijeti odluka o sljedećem grananju. Nema više nepridijeljenih vrijednosti. Pronađen je model i procedura je završila.

Sažetak PBO-a

- Ako dođe do konflikta:
  - Samo se odmota stog pridruživanja: Novi trenutačni stog pridruživanja = {poč. stanje stoga pridruživanja prije odluke o grananju}
  - Ne mijenjaju se promatrani literali (koji su vjerojatno mijenjani tijekom PBO) u bazi klauzula
  - Potrebno je samo promijeniti vrijednost varijable koja je dovela do konflikta kroz PBO, što se može napraviti u konstantnom vremenu
  - Nadalje, promjenom vrijednosti varijable koja je dovela do konflikta kroz PBO djelovat će samo na manji podskup klauzula od onoga u prvom pridruživanju, jer su **promatrani literali na nekim mjestima mijenjani**

- Ukupni je cilj minimizirati pristup klauzulama, jer je to pristupanje vremenski skupo

Heuristika odluke o varijabli grananja - **VSIDS**

- Engl. ***V**ariable **S**tate **I**ndependent **D**ecaying **S**um*
  - Varijable se rangiraju prema broju prisutnih literala (negiranih i nenegiranih) u **inicijalnoj** bazi klauzula
  - Ideja je da se literali koji se češće pojavljuju u novijim konfliktnim klauzulama najprije odabiru čime se lokalizira pretraga; a to se postiže:
    - Broj prisutnih literala za neku varijablu se inkrementira (za 1, tzv. *bump*) samo u slučaju dodavanja novih klauzula
    - Periodično, pomnože se svi brojevi s nekom konstantom $x < 1$ (raspad, engl. *decay*)
  - Pri odluci o grananju, odabire se prva po rangu (nepridijeljena) varijabla (i to njezin češći literal), a u slučaju izjednačenja, literal se odabere slučajno
  - **Odabrani literal postavlja se na FALSE radi smanjenja veličine klauzule**

VSIDS primjer

Inicijalna baza klauzula

$x1 \vee x4$
$x1 \vee \neg x3 \vee \neg x8$
$x1 \vee x8 \vee x12$
$x2 \vee x11$
$\neg x7 \vee \neg x3 \vee x9$
$\neg x7 \vee x8 \vee \neg x9$
$x7 \vee x8 \vee x10$

Rezultati:
4: x8
3: x1,x7
2: x3
1: x2,x4,x9,x10,x11,x12

Za grananje se odabere literal x8 (ne ¬x8), jer je najčešći (x8 – 3, ¬x8 – 1)

Dodana klauzula

$x1 \vee x4$
$x1 \vee \neg x3 \vee \neg x8$
$x1 \vee x8 \vee x12$
$x2 \vee x11$
$\neg x7 \vee \neg x3 \vee x9$
$\neg x7 \vee x8 \vee \neg x9$
$x7 \vee x8 \vee x10$
$x7 \vee x10 \vee \neg x12$

Rezultati:
4: x8, x7
3: x1
2: x3, x10, x12
1: x2,x4,x9,x11

Za grananje se opet odabere x8, jer je taj literal češći od x7 (3 : 2)

VSIDS primjer

Množenje s npr. 0.5 (raspad), zaokruži prema dolje

$x1 \vee x4$
$x1 \vee \neg x3 \vee \neg x8$
$x1 \vee x8 \vee x12$
$x2 \vee x11$
$\neg x7 \vee \neg x3 \vee x9$
$\neg x7 \vee x8 \vee \neg x9$
$x7 \vee x8 \vee x10$
$x7 \vee x10 \vee \neg x12$

Rezultati:
2: x8, x7
1: x3, x10, x12, x1
0: x2,x4,x9,x11

Za grananje se opet odabere x8, jer je taj literal češći od x7 (3 : 2)

Dodana nova klauzula

$x1 \vee x4$
$x1 \vee \neg x3 \vee \neg x8$
$x1 \vee x8 \vee x12$
$x2 \vee x11$
$\neg x7 \vee \neg x3 \vee x9$
$\neg x7 \vee x8 \vee \neg x9$
$x7 \vee x8 \vee x10$
$x7 \vee x10 \vee \neg x12$
$\neg x12 \vee x10$

Rezultati:
2: x8, x7, x10, x12
1: x3, x1
0: x2,x4,x9,x11

Sada bi se slučajno izabralo između x8 i x10 pri grananju (oba literala se pojavljuju 3 puta)

Interakcija PBO – VSIDS

- PBO obrađuje pridruživanja, čime neki literali prestaju biti promatrani
- Rang kod VSIDS-a se relativno sporo mijenja
- Općenito, što je varijabla više rangirana po VSIDS-u, to je vjerojatnije da više nije promatrana (jer ju je PBO zamijenio s nekom drugom)
- VSIDS utječe na to koje varijable će se naći u novo naučenim klauzulama proizašlima iz konflikta
- Bitno je zbog brzine da VSIDS održava strogo lokalizirane pretrage (pogotovo u slučaju velikog broja varijabli)

Brisanje naučenih klauzula

- Da bi se spriječio problem eksplozije zauzetog memorijskog prostora koji se može dogoditi učenjem novih klauzula nakon konflikta, uvodi se brisanje nekih naučenih klauzula
- Chaff koristi **planirano lijeno uklanjanje** (engl. *scheduled lazy deletion*)
  - **U trenutku dodavanja** klauzule, za nju se ocijeni kada (i da li uopće) će se izbrisati
  - Koristi se **faktor relevantnosti klauzule *N*** (tipično 100 – 200) koji kaže da kad prvi put više od *N* literala u klauzuli postane nepridijeljeno (odmatanjem stoga pridruživanja) da će se klauzula označiti za uklanjanje

Pretraga iznova (*restart*)

- Služi za rješavanje teških problema (bilo SAT, bilo ne-SAT)
- Napuštanje trenutačnog stanja pretrage i početak nove pretrage, uz zadržavanje naučenih (konfliktnih) klauzula da se ne bi ponovila analiza
- **Dodaje se mali šum pri prvih nekoliko odluka o grananju kako bi se istražio neki drugi put pretraživanja**
- I šum i učestalost pretrage iznova se mogu konfigurirati; po defaultu, učestalost se usporava sa svakom novom pretragom iznova

- Zbog pretrage iznova i brisanja nekih naučenih klauzula, teoretski se može dogoditi da pretraga ne bude kompletna. Kompletnost pretrage se ipak zadržava tako što se faktor relevantnosti klauzule *N* povećava s vremenom
- Kompletnost je uvijek održana ako se isključi pretraga iznova, no to može značajno produžiti računanje kod teških problema

Algoritam MiniSAT (N. Eén, N. Sörensson, 2003.-2008.)

- Minimalan ☺, dobro dokumentiran i učinkovit SAT-rješavač
- Manje razlike u odnosu na Chaff:
  - Najveći fokus na **učinkovitoj analizi konfliktnih klauzula**
    - Minimizira se veličina naučene klauzule detaljnijim razmatranjem klauzula koje su dovele do konflikta
    - Konfliktna klauzula se odmotava unazad primjenom **rezolucijskog pravila** na zadnjem propagiranom literalu
    - Staje se kada nova klauzula sadržava točno jedan novi literal zaključen od trenutka zadnje pretpostavke (grananja)
  - MiniSAT **ne razlikuje polarnosti literala** kod VSIDS mjere – uvijek pridjeljuje vrijednost *false* najviše rangiranoj varijabli

Algoritam MiniSAT (N. Eén, N. Sörensson, 2003.-2008.)

- Rezolucijsko pravilo (J. A. Robinson): Logička posljedica dviju istinitih, konjunkcijom vezanih, univerzalno kvantificiranih normaliziranih klauzula (disjunkcija literala) je normalizirana klauzula **bez jednog komplementarnog para literala.**

1. $(A_1 \vee A_2 \vee \ldots \vee A_n \vee \mathbf{B})$
2. $((\neg \mathbf{B}) \vee C_1 \vee C_2 \vee \ldots \vee C_p)$

---

3. $(A_1 \vee A_2 \vee \ldots \vee A_n \vee C_1 \vee C_2 \vee \ldots \vee C_p)$

Naprimjer, rezolucijsko pravilo nad varijablom a:

ako $(a \vee b)$ i $(\neg a \vee c \vee d)$ **onda zaključi** $(b \vee c \vee d)$

Primjer analize konflikta

Antecedent	**Pridruživanje**
pretpostavka	e=F
$\neg f \vee e$	f=F
$\neg g \vee f$	g=F
$\neg h \vee g$	h=F
pretpostavka	a=T (nova razina)
$b \vee \neg a \vee e$	b=T
$c \vee e \vee f$	c=T
$d \vee \neg b \vee h$	d=T

Npr. konflikt klauzule $d \vee \neg b \vee h$ s klauzulom: $\neg b \vee \neg c \vee \neg d$

Rezolucija 1 (po d):
$\neg b \vee \neg c \vee h$

Rezolucija 2 (po c):
$\neg b \vee h \vee e \vee f$

Tu stajemo, b je ostala jedina varijabla od trenutka zadnje pretpostavke (a=T), naučili smo novu konfliktnu klauzulu:
$\neg b \vee h \vee e \vee f$

Daljnja minimizacija veličine naučene klauzule

Antecedent	Pridruživanje
pretpostavka	e=F
$\neg f \vee e$	f=F
$\neg g \vee f$	g=F
$\neg h \vee g$	h=F
pretpostavka	a=T (nova razina)
$b \vee \neg a \vee e$	b=T
$c \vee e \vee f$	c=T
$d \vee \neg b \vee h$	d=T

Za daljnju minimizaciju veličine naučene klauzule može se "pohlepno" primijeniti rezolucijsko pravilo dokle ide, čak i iza razine posljednje pretpostavke

To MiniSAT ne radi

$\neg b \vee \mathbf{h} \vee e \vee f$

$\neg b \vee \mathbf{g} \vee e \vee f$

$\neg b \vee e \vee \mathbf{f}$

$\neg b \vee e$

Gdje stati?
(nije uvijek manja naučena klauzula bolja)

Predobrada i unutarnja obrada

- U novije vrijeme veliki naglasak pri izradi učinkovitog SAT-rješavača igraju metode za predobradu CNF-formule (engl. *SAT preprocessing*) i metode unutarnje obrade (engl. *SAT inprocessing*)
  - Neki put je diskutabilno može li se predobrada i unutarnja obrada smatrati dijelom samog algoritma za SAT-rješavanje
- Primjer *SAT preprocessing* metoda:
  - PJK, UKČL, ograničena eliminacija varijabli (engl. *bounded variable elimination*), SAT pometanje (engl. *SAT sweeping*)
- Primjer *SAT inprocessing* metoda: rezolucijsko pravilo, proširena rezolucija (engl. *clause-constrained extended resolution*)

Više u: A. Biere, M. Heule, H. Van Marren, T. Walsh, Handbook of Satisfiability, 2nd ed., IOP Press, 2021.

Predobrada i unutarnja obrada

- Ograničena eliminacija varijabli zasnovana je na distribuciji klauzula koja se provodi rezolucijom
  - Izabire se varijabla, dodaju se svi rezolventi te varijable u formulu i uklone se originalne klauzule koje ju sadrže
- Primjer:

$$F = (x \vee e) \wedge (y \vee e) \wedge (\bar{x} \vee z \vee \bar{e}) \wedge (y \vee \bar{e}) \wedge (y \vee z)$$

- Neka za distribuciju klauzula odaberemo varijablu e, rezoluciju na prve dvije klauzule možemo napraviti s trećom i četvrtom, dobivaju se četiri dodatne klauzule:

$$F \wedge (x \vee \bar{x} \vee z) \wedge (x \vee y) \wedge (y \vee \bar{x} \vee z) \wedge (y)$$

- I konačno, miču se klauzule koje sadržavaju e:

$$(y \vee z) \wedge (x \vee \bar{x} \vee z) \wedge (x \vee y) \wedge (y \vee \bar{x} \vee z) \wedge (y)$$

Neki najnoviji uspješni (*state-of-the-art*) SAT-rješavači

- Glucose SAT-solver (2009.+, paralelizacija 2014., L. Simon)
  - Optimizacija procesa upravljanja naučenim klauzulama – agresivnije, dinamičko uklanjanje koje štiti one naučene klauzule koje se češće koriste pri PBO
  - University of Bordeaux - http://www.labri.fr/perso/lsimon/glucose/
- Lingeling, Plingeling, Treengeling (2010.+, A. Biere)
  - Optimizacija niza parametara algoritama, fokus na proučavanju kako najbolje podesiti pretragu iznova, paralelizacija rješavanja SAT-problema
  - Johannes Kepler University of Linz - http://fmv.jku.at/lingeling/
- MapleSAT (i slični) (2016.+, J. H. Liang)
  - Razmatranje grananja kao optimizacijskog problema s ciljem maksimizacije učenja novih klauzula, heuristika LRB je brža od VSIDS-a, ograničena el. varijabli...
  - University of Waterloo i suradnici - https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/
- Rezultati posljednjeg natjecanja SAT-rješavača:
  - https://satcompetition.github.io/2022/results.html

PRIMJENA SAT-RJEŠAVAČA U FORMALNOJ VERIFIKACIJI

Primjena SAT-rješavača u provjeri modela

- SAT-rješavači se u formalnoj verifikaciji najviše koriste u ograničenoj provjeri modela (engl. *bounded model checking*, BMC)
  - Tipično za verifikaciju obilježja sigurnosti: možemo li pronaći loše stanje u *k* koraka?
  - Ili: možemo li pronaći protuprimjer u *k* koraka?
  - Koristi se uglavnom LTL-logika, ali i CTL je podržan
- *k* – koraka: ograničenje na kratak put izvođenja kako bi propozicijska formula bila dovoljno jednostavna za pronaći model
- Problem: što ako se protuprimjer dogodi, ali nakon *k* koraka? **Kod BMC-a, ne otkrije ga se!** Ideja je da se istražuju sve duži i duži putevi, ako se ima dovoljno vremena.

BMC i provjera modela – ideja

- Zadana je Kripkeova struktura M i vremenska formula φ
- Izgradimo formulu $\Gamma(k)$ koja je zadovoljiva akko φ vrijedi na putu duljine *k* stanja
- Put duljine *k* stanja: $I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$
- Za φ = **EF** *p* i za *k* = 2:

$$\Gamma(2) = I(s_0) \wedge R(s_0, s_1) \wedge R(s_1, s_2) \wedge (p_0 \vee p_1 \vee p_2)$$

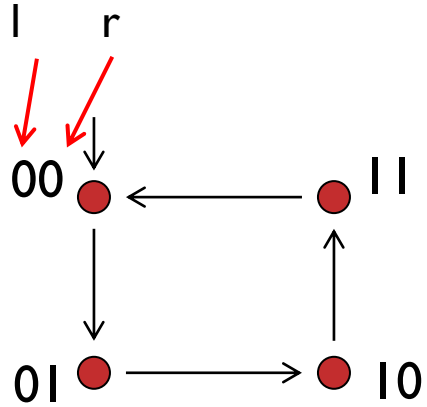
- Za φ = **AG** *p*:

$$\neg\Gamma(k) = (I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})) \rightarrow \bigwedge_{i=0}^{k} p_i$$, odnosno

$$\Gamma(k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p_i$$ Tražimo protuprimjere!

p je očuvan do *k*-tog prijelaza akko je Γ nezadovoljiva

Primjer: Dvobitni brojač




```
MODULE main
VAR
r : boolean;
l : boolean;
```

```
ASSIGN
init(r) := FALSE;
init(l) := FALSE;
next(r) := !r;
next(l) := r xor l;
```

- Početno stanje: $I : \neg r \wedge \neg l$
- Relacija prijelaza: R: $\begin{pmatrix} l' = (l \neq r) \wedge \\ r' = \neg r \end{pmatrix}$
- Obilježje sigurnosti: **AG** $(\neg l \vee \neg r)$

Ovo se prevede u CNF formu

$$\Gamma(2) : (\neg l_0 \wedge \neg r_0) \wedge \begin{pmatrix} l_1 = (l_0 \neq r_0) \wedge r_1 = \neg r_0 \\ l_2 = (l_1 \neq r_1) \wedge r_2 = \neg r_1 \end{pmatrix} \wedge \begin{pmatrix} (l_0 \wedge r_0) \vee \\ (l_1 \wedge r_1) \vee \\ (l_2 \wedge r_2) \end{pmatrix}$$

$\Gamma(2)$ nije zadovoljiv, ali $\Gamma(3)$ bi bio zadovoljiv, što ne bi pronašli za k = 2.

Ulazni format DIMACS CNF

- Standardni ulazni format datoteke za rješavanje SAT-problema
- ASCII format, ekstenzija: .cnf
- *Benchmark* skupovi za SAT: http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html
- Opis formata:

```
c
c start with comments
c
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

Komentari počinju s "c", obično na početku datoteke

Problemska linija počinje s "p" , zatim se navodi tip problema (cnf, wcnf, dnf), broj varijabli i na kraju broj klauzula

Klauzule, svaka u svojem retku, završavaju s "0".

Minus ispred broja varijable označava njenu negaciju.

Daljnje učenje – MAX-SAT

- MAX-SAT – problem određivanja najvećeg broja klauzula neke Booleove formule u CNF-obliku koje mogu biti zadovoljive za neko pridruživanje istinitosti
  - NP-težak problem, teži od "običnog" SAT problema
- Značajnija proširenja MAX-SAT problema:
  - Težinski MAX-SAT (engl. *weighted MAX-SAT*) – svaka klauzula ima zadanu težinu – traženje maksimalne težine skupa zadovoljivih klauzula
  - Djelomični MAX-SAT (engl. *partial MAX-SAT*) – za dani podskup klauzula radi se MAX-SAT, ostale klauzule moraju biti SAT
- Natjecanje MAX-SAT-rješavača: https://maxsat-evaluations.github.io/2022/
- Primjer MAX-SAT-rješavača: QMaxSAT – težinski, djelomični MAX-SAT rješavača ne garantira kompletnost – korisno za optimizacijske probleme
  - https://sites.google.com/site/qmaxsat

Daljnje učenje: SMT-rješavači

- Dokazivanje teorija (cjelobrojnih formula, realnih formula, bit-vektorskih formula, kvantifikacija varijabli...) koje su zadane u nekom obliku logike prvog reda s jednakosti
- U nekim slučajevima koriste pretvorbu problema u SAT-oblik pa dalje koriste SAT-rješavače
- Koristi se za automatsku verifikaciju programa (npr. provjeru ugovora, simboličko izvršavanje programa) s ciljem pronalaska pogrešaka, a također i za formalnu sintezu
- Najpoznatiji primjeri SMT-rješavača:
  - Z3: https://github.com/Z3Prover/z3/wiki
  - Yices: http://yices.csl.sri.com/

BDDs vs SAT vs …

- Svi pristupi verifikaciji su komplementarni
- SAT-rješavači ne zamjenjuje BDD-ove, niti QBF-rješavači ili SMT-rješavači ne zamjenjuje SAT-rješavače / BDD-ove
- **"*No silver bullet*" – nema postupka koji će na svim problemima dati najbolji rezultat** – sve ovisi o vrsti problema – neki problemi se bolje rješavaju s BDD-om, a neki sa SAT-om ili SMT-om
- Nažalost, često u literaturi nije unaprijed poznato koji postupak koristiti – treba isprobati više njih
- Nekih pravila ima: BMC sa SAT-om je brži za pronalaženje "plićih" pogrešaka i za davanje kraćih protuprimjera od BDD-ova
- Postupci zasnovani na BDD-ovima su obično bolji od SAT-a za dokazivanje odsutnosti pogreške jer jednostavnije rade nad beskonačnim putovima kroz sustav

Zadaci

1. Pokažite zadovoljivost zadane formule u CNF-obliku korištenjem temeljnog DPLL rješavača. Izbor varijable grananja, ako je on potreban, provedite proizvoljno.

$\Gamma = (x1 \vee \neg x2 \vee x3) \wedge (\neg x1 \vee \neg x2 \vee x3) \wedge (x1) \wedge (x1 \vee \neg x2) \wedge (\neg x2 \vee \neg x4) \wedge (\neg x1 \vee x2 \vee x4)$

2. Imate na raspolaganju SAT-rješavač GRASP.

Za zadani početni skup klauzula K_1-K_9

i za trenutno pridruživanje

$\{x_3=0@1, x_7=0@2, x_{10}=0@2, \ldots\}$:

a) Nacrtajte graf implikacija ako je

trenutna odluka o pridruživanju $x_2 = 1@4$

b) Odredite naučene konfliktne klauzule za

otkrivene konflikte na kraju grafova implikacija

c) Odredite jedinstvenu implikacijsku točku

za konflikte prouzročene varijablom x_2 te razinu

odluke δ na koju će algoritam skočiti nakon konflikata

$K_1 = (\neg x_1 \vee x_5 \vee x_9)$
$K_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_4)$
$K_3 = (x_1 \vee \neg x_2 \vee x_3)$
$K_4 = (x_5 \vee x_2 \vee x_3)$
$K_5 = (\neg x_5 \vee x_8)$
$K_6 = (x_4 \vee x_8)$
$K_7 = (x_7 \vee \neg x_8 \vee \neg x_9)$
$K_8 = (\neg x_1 \vee \neg x_2 \vee \neg x_5)$
$K_9 = (x_2 \vee \neg x_5 \vee \neg x_8)$
...

Zadaci

3. Pokažite zadovoljivost zadanog skupa klauzula korištenjem Chaff-rješavača. Pritom koristite sve značajke algoritma osim uklanjanja naučenih klauzula i pretrage iznova.

$K_1 = (\neg x_1 \vee x_2 \vee x_4)$

$K_2 = (\neg x_1 \vee x_2 \vee \neg x_4)$

$K_3 = (\neg x_1 \vee \neg x_2 \vee x_4)$

$K_4 = (\neg x_1 \vee \neg x_2 \vee \neg x_4)$

$K_5 = (x_1 \vee x_3)$

$K_6 = (x_1 \vee \neg x_3)$

FVPP: SMT-rješavači i simbolička verifikacija programa

Verifikacija složenijih logičkih sustava

Pripremio: izv. prof. dr. sc. Alan Jović

Ak. god. 2022./2023.

Sadržaj

- Uvod u SMT-rješavače
- Teorije
- Primjer primjene i veza sa SAT-rješavačima
- Uvod u simboličku verifikaciju programa

UVOD U SMT-RJEŠAVAČE

SMT-rješavači

- **SMT-rješavač (engl. *Satisfiability Modulo Theory, SMT solver*)** je računalni program koji **odlučuje o zadovoljivosti formule predikatne logike prvoga reda** (FOPL) uz dodatak **teorija** kao što su teorija jednakosti, cijelih i realnih brojeva, polja, itd.
  - Teorije koje rješava SMT-rješavač su puno složenije od onih koje rješava SAT-rješavač
- Primjena SMT-rješavača je u raznim područjima znanosti i tehnike, a posebice u:
  - **formalnoj verifikaciji** – automatsko dokazivanje teorema
  - **simboličkom izvršavanju programa** – analiza i verifikacija rada programa

SMT-rješavači

- Problemi koje SMT-rješavači rješavaju mogu imati više stotina pa i tisuća klauzula kao što je ova:

  $p \vee \forall x P(x) \vee a=f(b-c) \vee g(g(b)) \neq c \vee (a-c) \leq 7$

- U programima, klauzule opisuju ograničenja nad varijablama, koja su zadana strukturom programa ili njegovim preduvjetima
- SMT-rješavači se smatraju programima koji učinkovito rješavaju probleme s ograničenjima pa ih se još naziva **rješavačima ograničenja** (engl. *constraint solver*)

SMT-rješavači

- Najčešće samostalni i brzi programi koje pozivaju drugi alati, npr. oni za simboličko izvršavanje programa ili oni koji rješavaju neke praktične probleme (npr. izrada rasporeda sati)
- Suvremeni SMT-rješavači uspješno rade s oko 100 000 Booleovih varijabli (i ekvivalentnim brojem ne-Booleovih varijabli) i s programima od oko 100 000 linija koda
- Primjeri ograničenja na koje SMT-rješavači mogu naići, od kojih neka ne moraju nužno znati riješiti:
  - Linearna ograničenja: $5x + 6 < 100$
  - Nelinearna ograničenja: $x \cdot y + 2 < 35$
  - Neinterpretirane funkcije: $f(x) < 4$
- Zadatak SMT-rješavača je da pronađe zadovoljavajuće pridruživanje vrijednosti varijabli uz zadana ograničenja

TEORIJE

Teorije

- **Teorija = gradivni dio logičke formule koja tvori ograničenje**
- Matematički jasno i precizno opisana logička struktura, zadana je svojim **potpisom** (domenom djelovanja) i **aksiomima** (tvrdnjama koje u njoj vrijede)
- Svaka teorija, kada se razmatra kao **problem odlučivanja**, može biti odlučljiva (*decidable*) ili neodlučljiva (*undecidable*)
- Napomena: iako odlučljive, neke teorije imaju visoku složenost odlučivanja (npr. eksponencijalnu)

Primjer – Teorija jednakosti T_E

- **Potpis** teorije jednakosti T_E :
  - Simboli za neinterpretirane funkcije (npr. f, g, ...), predikate (npr. P, Q, ...), varijable (npr. x, y, z...) i konstante (npr. a, b, ...)
  - Binarni predikat "=", interpretiran sa značenjem koje mu daju aksiomi
- **Aksiomi** teorije jednakosti T_E :

1. $\forall x.\ \ x = x$ (refleksivnost)
2. $\forall x,y.\ x = y \rightarrow y = x$ (simetrija)
3. $\forall x,y,z.\ x = y \wedge y = z \rightarrow x = z$ (tranzitivnost)
4. Za svaki pozitivni cijeli broj *n* i *n*-arni funkcijski simbol *f*,

   $\forall x_1,\ldots,x_n, \forall y_1,\ldots,y_n.\ x_1 = y_1 \wedge \ldots x_n = y_n \rightarrow f(x_1,\ldots,x_n) = f(y_1,\ldots y_n)$

   (kongruentnost funkcija)
5. Za svaki pozitivni cijeli broj *n* i *n*-arni predikatni simbol *P*,

   $\forall x_1,\ldots,x_n, \forall y_1,\ldots,y_n.\ x_1 = y_1 \wedge \ldots x_n = y_n \rightarrow P(x_1,\ldots,x_n) = P(y_1,\ldots y_n)$

   (kongruentnost predikata)

Primjer – Teorija jednakosti T_E

Primjeri:

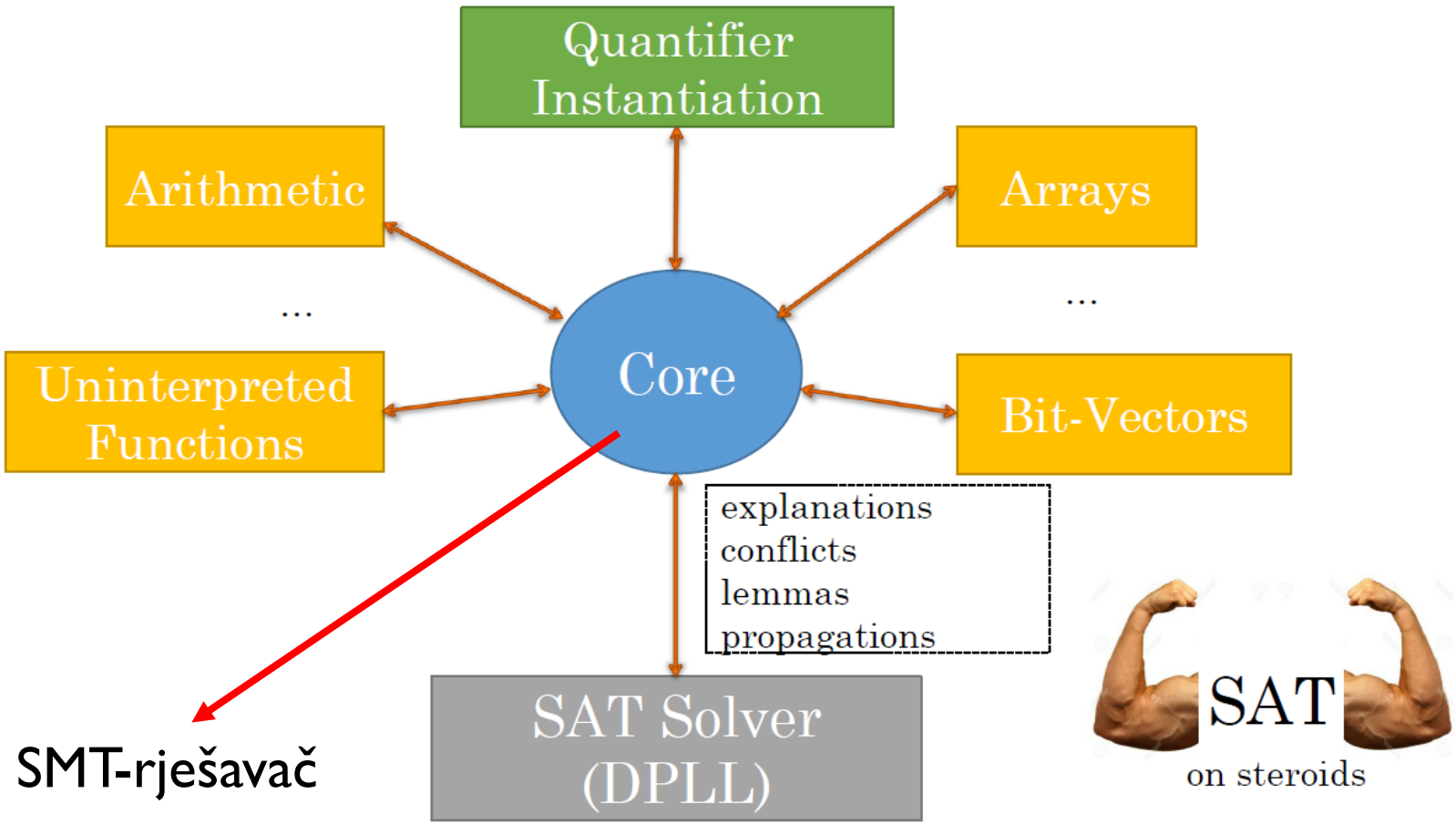
- za binarni funkcijski simbol *f* i *n* = 2 vrijedi tvrdnja:

$$\forall x_1, x_2, y_1, y_2 .\ x_1 = y_1 \wedge x_2 = y_2 \rightarrow f(x_1, x_2) = f(y_1, y_2)$$

- vrijedi tvrdnja:

$$a = b \wedge b = c \rightarrow g(f(a), b) = g(f(c), a)$$

SMT-rješavači i podržane teorije



SMT-rješavač

Teorije – odlučljivost

Theory	Description	Full	QFF
T_{E}	equality	no	yes
T_{PA}	Peano arithmetic	no	no
$T_{\mathbb{N}}$	Presburger arithmetic	yes	yes
$T_{\mathbb{Z}}$	linear integers	yes	yes
$T_{\mathbb{R}}$	reals (with $\cdot$)	yes	yes
$T_{\mathbb{Q}}$	rationals (without $\cdot$)	yes	yes
T_{RDS}	recursive data structures	no	yes
T^{+}_{RDS}	acyclic recursive data structures	yes	yes
T_{A}	arrays	no	yes
$T^{=}_{\mathsf{A}}$	arrays with extensionality	no	yes

- QFF – formula bez kvantifikatora – fragment teorije – npr. $\mathrm{P(f(a), b)} \wedge \mathrm{c = g(d)}$
- T_{PA} – 0, 1, +, *, =
- T_N – prirodni brojevi, +, =
- T_Z – cijeli brojevi,+, -, >, =, množenje konstantom
- …
- Nešto detaljnije o odlučljivosti pojedinih teorija može se pročitati ovdje: http://web.stanford.edu/class/cs156/0910/slides/coc_technion_3.pdf

Model (semantika) teorije i pokazivanje ispravnosti

- Model M neke teorije T je definiran kao:
  - **Domena S** – skup elemenata
  - **Interpretacija funkcija f** za svaki f iz skupa funkcijskih simbola s brojem argumenata n: $f^M: S^n \rightarrow S$
  - **Interpretacija predikata P** za svaki P iz skupa predikatnih simbola s brojem argumenata n: $P^M \subseteq S^n$
  - **Pridruživanje** $x^M \in S$ za svaku varijablu x iz skupa varijabli

- M je model za formulu φ ako ju model evaluira u istinitu za dane interpretacije na domeni S
- M je model za cijelu teoriju T ako su sve formule iz T istinite za M (ako sve formule imaju model M)

Model (semantika) teorije i pokazivanje ispravnosti

- Formula $\varphi(x)$ je **zadovoljiva** s obzirom na teoriju T ako postoji barem jedan **model** M za T koji formulu $\varphi(x)$ evaluira u istinu.
  - Notacija: $M \vDash_T \varphi(x)$
- Formula $\varphi(x)$ je **valjana (ispravna, tautologija)** s obzirom na teoriju T ako se $\varphi(x)$ evaluira u istinu za svaki model M od T
- Alternativno: Formula $\varphi(x)$ je valjana ako je $\neg\varphi(x)$ nezadovoljiva
- Drugim riječima: ako je $\mathcal{A}$ skup aksioma koji opisuju neku teoriju T, **tada je formula $\varphi(x)$ valjana s obzirom na $\mathcal{A}$ ako je** $\mathcal{A} \cup \neg\varphi(x)$ nezadovoljiva
- Primijetiti: teorija T u ovoj definiciji može biti po volji složena teorija
- Korak prema formalnoj verifikaciji programa:
  - Da bismo provjerili je li formula `pre` $\rightarrow$ `program` $\rightarrow$ `post` ispravna, tu formulu trebamo negirati i pozvati SMT-rješavač koji treba pokazati da je ta negacija nezadovoljiva (UNSAT)

Primjer – model za formulu iz T_E

- Provjerimo je li tvrdnja:

  $\varphi \equiv a = b \wedge b = c \rightarrow g(f(a), b) = g(f(c), a)$ valjana u T_E

- Neka je M neki model od T_E. M **ne bi bio model** od φ jedino ako je $M \models_T$ $a = b \wedge b = c$ i $M \not\models g(f(a), b) = g(f(c), a)$ (zbog implikacije $\rightarrow$)

- $M \models \forall x, y, z \,.\, x = y \wedge y = z \wedge x = z$ *(tranzitivnost)*

  $M \models a = c$

- $M \models \forall x, y.\ x = y \rightarrow f(x) = f(y)$ *(kong. funkcija)*

  $M \models f(a) = f(c)$

- $M \models a = b \wedge b = c$ *(elim. konjunkcije)*

  $M \models a = b$

Primjer – model za formulu iz T_E

- $M \models \forall x, y \,.\, x = y \wedge y = x$ *(komutativnost)*
  $M \models b = a$

- $M \models \forall x,y,u,v.\ x=y \wedge u=v \rightarrow g(x,u) = g(y,v)$ *(kongruentnost funkcija)*

- $M \models f(a) = f(c)$
- $M \models b = a$
- $M \models g(f(a), b) = g(f(c), a)$

- Došli smo do kontradikcije s pretpostavkom kada *M* ne bi bio model
- Zaključujemo: *M* je model od φ

Primjer – Peano aritmetika T_{PA}

- **Potpis**: konstante **0**, **1**, funkcije **+** i · te jednakost **=**
- Definirani su aksiomi za jednakost: refleksivnost, simetričnost, tranzitivnost, funkcije + i ·
- Dodatno, definirani su **aksiomi**:

1.	$\forall x.\ \neg(x + 1 = 0)$	(nula)
2.	$\forall x,y.\ x + 1 = y + 1 \rightarrow x = y$	(sljedbenik)
3.	$F[0] \wedge (\forall x.\ F[x] \rightarrow F[x+1]) \rightarrow \forall x.\ F[x]$	(indukcija)
4.	$\forall x.\ x + 0 = x$	(plus nula)
5.	$\forall x,y.\ x + (y + 1) = (x + y) + 1$	(plus sljedbenik)
6.	$\forall x.\ x \cdot 0 = 0$	(puta nula)
7.	$\forall x,y.\ x \cdot (y + 1) = x \cdot y + x$	(puta sljedbenik)

Primjer – Peano aritmetika T_{PA}

- Primjer: $3x + 5 = 2y$ može se zapisati u T_{PA} kao:

  $x + x + x + 1 + 1 + 1 + 1 + 1 = y + y$

  Napomena: mogli smo zapisati i

  $3x + 5 \geq 2y$ i to kao $\exists z.\ 3x + 5 = 2y + z$

- $\mathbf{T_{PA}}$ je **neodlučljiv** – s kvantifikacijom (Gödel, Tarski, Church, Turing, 1930-te) ili bez kvantifikacije varijable (Matiyasevich, 1970.),
- Fragment teorije $\mathbf{T_{PA}}$ **bez multiplikacije (·) je odlučljiv**

Primjer – Teorija polja T_A

- **Potpis**: funkcija ***read***; funkcija ***write***; jednakost **=** (samo za elemente polja, ne cijela polja)
  - Funkcija *read(a, i)* je binarna funkcija koja čita iz polja *a* na indeksu *i*
  - Funkcija *write(a, i, v)* je ternarna funkcija: zapisuje vrijednost *v* na indeksu *i* polja *a*
- **Aksiomi:**
- $\forall a,i,j.\ i = j \rightarrow read(a,\ i)\ = read(a,\ j)$ (kongruentnost polja)
- $\forall a,v,i,j.\ i = j \rightarrow read(write(a,\ i,\ v),\ j) = v$ (read –write 1)
- $\forall a,v,i,j.\ i \neq j \rightarrow read(write(a,\ i,\ v),\ j) = read(a,\ j)$ (read –write 2)
- `a[i] = v` iz programa se prevodi u `a = write(a, i, v)`
- Loša vijest: T_A je **neodlučljiva**
- Dobra vijest: Fragment teorije T_A bez kvantifikatora je **odlučljiv**

Kombiniranje teorija

- Neki problemi zahtijevaju kombiniranje više teorija
- Problem kombiniranja više teorija je težak
  - Je li kombinacija dviju odlučljivih teorija opet odlučljiva teorija?
  - Što napraviti kada u programu imamo ograničenja koja kombiniraju više teorija?
- **Općeg rješenja nema**!
- Postoje korisni posebni slučajevi rješenja

Primjer: algoritam binarne pretrage treba kombinaciju teorija **aritmetike cijelih brojeva T_Z** i **polja T_A (*arrays*) –** rješivo za suvremene SMT-rješavače

```
int binary_search(
    int[] arr, int low, int high, int key) {
        assert (low > high || 0 <= low < high);
        while (low <= high) {
            //Find middle value
            int mid = (low + high)/2;
            assert (0 <= mid < high);
            int val = arr[mid];
            //Refine range
            if (key == val) return mid;
            if (val > key) low = mid+1;
            else high = mid-1;
        }
        return -1;
}
```

Kombiniranje teorija

- Kako pokazati da je formula:

$$1 \leq x \;\wedge\; x \leq 2 \;\wedge\; f(x) \neq f(1) \;\wedge\; f(x) \neq f(2)$$

  ispravna u teoriji $(T_E \cup T_Z)$?

- Pokazuje se da ako postoje teorije T_1 i T_2 takve da im je presjek potpisa jednak {=}, onda zajednička teorija $(T_1 \cup T_2)$ ima potpis jednak uniji pojedinačnih potpisa i aksiome jednake uniji pojedinačnih aksioma.
- Nelson i Oppen su pokazali da ako je T_1 bez kvantifikatora odlučljiva, T_2 bez kvantifikatora odlučljiva i određeni jednostavni tehnički zahtjevi su ispunjeni, onda je i $(T_1 \cup T_2)$ bez kvantifikatora odlučljiva teorija

- Detalji: https://web.stanford.edu/class/cs357/lecture11.pdf

PRIMJER PRIMJENE I VEZA SA SAT-RJEŠAVAČIMA

Primjer: SMT-rješavač za raspoređivanje poslova koristeći aritmetiku razlike

- **Klasični problem raspoređivanja poslova** (engl. *job-scheduling*):
  - *n* poslova
  - Svaki posao sastoji se od *m* zadataka, svaki sa svojim trajanjem, koji se moraju slijedno izvoditi na *m* strojeva
  - Zadatak ne može biti prekinut jednom kada počne
- Ograničenja (2 vrste):
  - Prethođenje: zadan je redoslijed zadataka unutar jednog posla
  - Resursi: dva različita zadatka koji zahtijevaju isti stroj ne mogu biti pokrenuti istovremeno
- Zadano je najveće vrijeme *max* dokad svi poslovi moraju završiti

- **Zadatak:** na temelju zadanih trajanja zadataka svih poslova, pronaći raspored (ako postoji), takav da je konačno vrijeme izvršavanja svih zadataka svih poslova manje ili jednako *max*

Problem raspoređivanja poslova

$d_{i,j}$	Machine 1	Machine 2
Job 1	2	1
Job 2	3	1
Job 3	2	3

max = 8

Solution

$t_{1,1} = 5, \quad t_{1,2} = 7,$

$t_{2,1} = 2, \quad t_{2,2} = 6,$

$t_{3,1} = 0, \quad t_{3,2} = 3$

Encoding

$$(t_{1,1} \geq 0) \wedge (t_{1,2} \geq t_{1,1} + 2) \wedge (t_{1,2} + 1 \leq 8) \wedge$$
$$(t_{2,1} \geq 0) \wedge (t_{2,2} \geq t_{2,1} + 3) \wedge (t_{2,2} + 1 \leq 8) \wedge$$
$$(t_{3,1} \geq 0) \wedge (t_{3,2} \geq t_{3,1} + 2) \wedge (t_{3,2} + 3 \leq 8) \wedge$$
$$((t_{1,1} \geq t_{2,1} + 3) \vee (t_{2,1} \geq t_{1,1} + 2)) \wedge$$
$$((t_{1,1} \geq t_{3,1} + 2) \vee (t_{3,1} \geq t_{1,1} + 2)) \wedge$$
$$((t_{2,1} \geq t_{3,1} + 2) \vee (t_{3,1} \geq t_{2,1} + 3)) \wedge$$
$$((t_{1,2} \geq t_{2,2} + 1) \vee (t_{2,2} \geq t_{1,2} + 1)) \wedge$$
$$((t_{1,2} \geq t_{3,2} + 3) \vee (t_{3,2} \geq t_{1,2} + 1)) \wedge$$
$$((t_{2,2} \geq t_{3,2} + 3) \vee (t_{3,2} \geq t_{2,2} + 1))$$

$d_{i,j}$ = trajanje *j*. zadatka *i*. posla

$t_{i,j}$ = početak *j*. zadatka *i*. posla

Kodiranje početnog i završnog vremena:

$t_{i,1} \geq 0$: početno vrijeme 1. zadatka za *i*. posao

$t_{i,m} + d_{i,m} \leq max$: završno vrijeme zadnjeg *m*. zadatka najviše jednako *max*

Kodiranje prethođenja:

$t_{i,j+1} \geq t_{i,j} + d_{i,j}$: novi zadatak počinje kad prethodni završi

Kodiranje resursa:

$(t_{i,j} \geq t_{i',j} + d_{i',j}) \vee (t_{i',j} \geq t_{i,j} + d_{i,j})$: dva zadatka *j* od različitih poslova *i* te *i'* se ne izvode na istom stroju u isto vrijeme

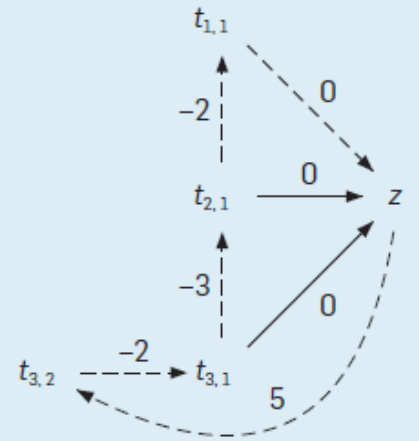
Teorija aritmetike razlike

- **Aritmetika razlike** (engl. *difference arithmetic*) podvrsta je **linearne aritmetike** T_Z **(linearna ograničenja nad cijelim brojevima)** u kojoj se dozvoljavaju samo formule oblika:

  $t - s \leq c$, gdje su t i s varijable, a c je numerička konstanta
- Ako su varijable u našem problemu zadane u obliku $s \leq c$ ili $c \leq s$ može ih se svesti na traženi oblik aritmetike razlike koristeći "nultu" varijablu $z = 0$, čime se dobiva: $s - z \leq c$ ili $z - s \leq -c$
- Aritmetiku razlike može se iskoristiti za riješiti klasični problem raspoređivanja poslova, budući da se sva ograničenja mogu prikazati u tom obliku
- Iz ograničenja u tom obliku oblikuje se **težinski usmjereni graf**

Težinski usmjereni graf – primjer

$$
\begin{array}{lcll}
z & - & t_{1,1} & \leq 0 \\
z & - & t_{2,1} & \leq 0 \\
z & - & t_{3,1} & \leq 0 \\
t_{3,2} & - & z & \leq 5 \\
t_{3,1} & - & t_{3,2} & \leq -2 \\
t_{2,1} & - & t_{3,1} & \leq -3 \\
t_{1,1} & - & t_{2,1} & \leq -2
\end{array}
$$



- Svaka varijabla je čvor, a nejednakost t - $s \leq c$ odgovara usmjerenom bridu od varijable s u varijablu t s težinom c
- Skup varijabli u aritmetici razlike može se učinkovito pretražiti za zadovoljivost tako da se traže **negativni ciklusi** u težinskom usmjerenom grafu.
- **Negativni ciklus je put kroz težinski usmjereni graf od čvora *x* nazad do čvora *x* na kojem je ukupan zbroj težina negativan**
- U ovom konkretnom primjeru prikazan je samo dio ukupnog težinskog usmjerenog grafa, gdje negativan ciklus čvora $t_{3,2}$ počinje putom -2 prema $t_{3,1}$ i nastavlja se po slijedu crtkanih linija; sve nazad to $t_{3,2}$, što odgovara rasporedu poslova posao1/zadatak1, posao2/zadatak1, posao3/zadatak1, posao3/zadatak2 koji **ne može završiti u *max* vremenu**

Pokazivanje zadovoljivosti – pristup "lazy offline"

- Problem ograničenja u aritmetici razlike preslikava se u SAT-problem tako da se uvedu **nove Booleove varijable**, npr. umjesto $\neg(a \geq 3) \wedge (a \geq 3 \vee a \geq 5)$ Booleova formula bi glasila $\neg p1 \wedge (p1 \vee p2)$.
- **Ako SAT-oblik pokaže nezadovoljivost, onda je i problem ograničenja nezadovoljiv, a ako pokaže zadovoljivost, onda se dalje iskoristi SMT-rješavač da se provjeri model od SAT-a**
  - Npr. ($p1$ = false, $p2$ = true) je model od SAT-a za gornji primjer, a kad se taj model prevede u SMT: ($\neg(a \geq 3)$, $a \geq 5$), to je nezadovoljivo u teoriji linearne aritmetike.

$p1$ $p2$

Pokazivanje zadovoljivosti – pristup "lazy offline"

- Budući da je SMT bio nezadovoljiv, dalje se negira prethodna formula: $(a \geq 3 \vee \neg(a \geq 5))$ čime ovo postaje valjana SMT formula i čija je apstrakcija u SAT-obliku $p1 \vee \neg p2$.
- Ovu izvedenu klauzulu zovemo "**lema teorije**".
- Budući da je ova lema SMT teorije valjana formula, može se dodati originalnoj formuli i dobiti $\neg p1 \wedge (p1 \vee p2) \wedge (p1 \vee \neg p2)$
- Ponovno se pokrene SAT-rješavač koji sada **ne pronalazi model**
- Ovime smo dokazali da je izvorna formula:

  $\neg(a \geq 3) \wedge (a \geq 3 \vee a \geq 5)$ nezadovoljiva.

Pokazivanje zadovoljivosti – pristup "lazy offline"

- U praksi, u nizu koraka stvara se puno lema teorije dok se ne dođe do konačnog odgovora o modelu koji može biti:
  - **SAT i SMT kažu da postoji model ili**
  - **SAT kaže da nema modela**
- **Rješenje se uvijek pronalazi jer imamo konačni broj varijabli i odlučljivu teoriju**
- U našem primjeru, negacija nezadovoljivog negativnog ciklusa u težinskom usmjerenom grafu bila bi valjana:

  $\neg(t_{3,1} - t_{3,2} \leq -2) \vee \neg(t_{2,1} - t_{3,1} \leq -3) \vee \neg(t_{1,1} - t_{2,1} \leq -2) \vee \neg(z - t_{1,1} \leq 0) \vee \neg(t_{3,2} - z \leq 5)$ i činila bi lemu teorije koja bi se onda dalje dokazivala...

Najuspješniji SMT-rješavači

- Svi navedeni podržavaju rješavanje većine odlučljivih teorija:
  - **Z3** (Microsoft) (2008.+) – https://github.com/Z3Prover/z3
  - **Yices 2** (SRI internation, NSF, NASA, DARPA) (2009.+) – http://yices.csl.sri.com/
  - CVC4 (Stanford University) (2013.+) – http://cvc4.cs.stanford.edu/web/
  - MathSAT 5 (University of Trento) (2013.+) – http://mathsat.fbk.eu/
  - Boolector (Johannes Kepler University, Austria) (2009.+) – https://boolector.github.io/
- Ulazni jezik: SMT-lib - https://smtlib.cs.uiowa.edu/
- Godišnja natjecanja SMT-rješavača: https://smt-comp.github.io

UVOD U SIMBOLIČKO IZVRŠAVANJE PROGRAMA

Cilj, zadatak i problemi...

- **Cilj:** automatsko pronalaženje kvarova u programu generiranjem ispitnih slučajeva koji ih prokazuju
- **Zadatak**: potrebno je proći kroz sve putove programa (idealno), ili kroz što veći broj putova u ograničenom vremenu (realno)
- Simboličko izvršavanje programa poznato je već skoro 50 godina (1975.+), ali se tek odnedavno uspješno ostvaruje, zbog snažnog razvoja SAT-rješavača i rješavača ograničenja (engl. *constraint solver*)
- Problemi koje simboličko izvršavanje želi riješiti i na kojima se radi:
  - Grananja pri izvršavanju uzrokuju eksponencijalni porast broja putova
  - Petlje generiraju vrlo veliki broj putova
  - Varijable programa mogu poprimiti veliki raspon vrijednosti i razne vrste ograničenja
  - Istovremeno izvršavanje je problematično za analizu i ispitivanje
  - Komunikacija s vanjskim knjižnicama, upravljanje memorijom...

Značajke simboličkog izvršavanja

- Ključne značajke simboličkog izvršavanja:
  1. **Za ulazne vrijednosti programa koriste se simboličke vrijednosti umjesto konkretnih vrijednosti**
  2. **Varijable programa prikazuju se kao simbolički izrazi nad simboličkim vrijednostima**
  3. **Pri ispitivanju programa, simboličko izvršavanje se koristi za generiranje konkretnih ulaznih podataka za svaki ostvarivi put kroz program**
- Tijekom izvođenja programa dodaju se nova ograničenja nad varijablama

Značajke simboličkog izvršavanja

- Simboličko izvršavanje programa održava u svakom trenutku:
  - σ - simboličko stanje (engl. *symbolic state, symbolic store*)
    - Preslikava dotad posjećene varijable programa u simboličke izraze
  - SPC - simboličko ograničenje puta (engl. *symbolic path constraint, symbolic path condition*)
    - Služi za pamćenje svih dotad posjećenih **grananja** u programu
    - Logička formula u odlučljivom obliku logike prvoga reda bez kvantifikatora
    - Razrješava se da bi se našle konkretne vrijednosti na kraju ostvarivog puta

Primjer simboličkog izvršavanja

Program:

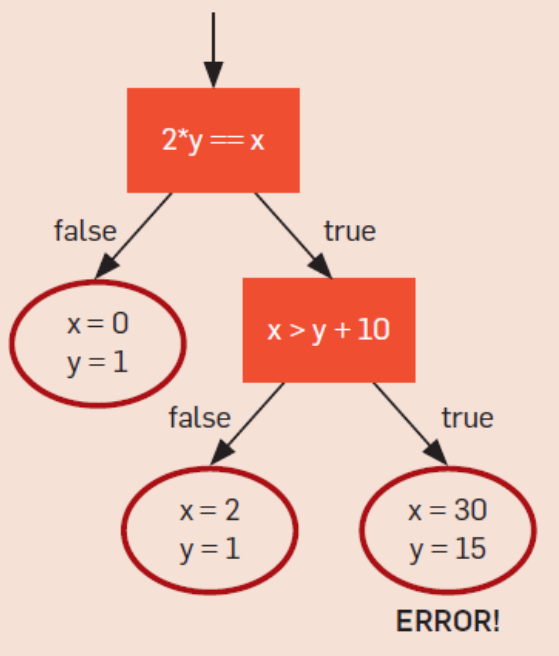
```
int twice(int v) {
  return 2 * v;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
        ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

- Zadatak je utvrditi koji ulazni podatci dovode izvođenje programa do retka "ERROR"
- U praksi, "ERROR" može biti bilo koja vrsta pogreške
- Budući da ne znamo *a priori* gdje se pogreška nalazi, trebamo ispitati sve putove

- Postoje tri moguća puta kroz program, primjer stabla izvršavanja:



Primjer simboličkog izvršavanja

Program:

```
int twice(int v) {
  return 2 * v;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
        ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

- Kako se odvija simboličko izvršavanje ovog programa?

1. Funkcije `read` dobavljaju ulazni podatak odnekud (npr. iz komandne linije), ne znamo njihovu vrijednost
2. Postavljaju se varijable *x* i *y* na nove simboličke vrijednosti

- Početno stanje simboličkog izvršavanja prije funkcije `testme(x,y)` izgleda ovako:

$\sigma : x \rightarrow x0$
$y \rightarrow y0$

SPC = true
(zasad još nema posjećenih grananja)

Primjer simboličkog izvršavanja

Program:

```
int twice(int v) {
  return 2 * v;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
        ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

3. Izvršava se funkcija `twice`, dobiva se novo stanje:

σ : x → x0 **SPC** = true
y → y0
z → 2*y0

4. Izvršavanje se dijeli na 2 puta (dupliciraju se stanja programa):

za (`x == z`):

σ : x → x0 **SPC** : x0 = 2*y0
y → y0
z → 2*y0

za (`x ≠ z`):

σ : x → x0 **SPC** : x0 ≠ 2*y0
y → y0
z → 2*y0

Moglo bi se stati s istraživanjem pojedinog puta ako bi se znalo da je SPC nezadovoljiv. U ovom slučaju, oba SPC-a su zadovoljiva pa se nastavlja.

Primjer simboličkog izvršavanja

Program:

```
int twice(int v) {
  return 2 * v;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
        ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

Neka se dalje istražuje put za (`x == z`) :

5. Dijeli se dalje na 2 puta (ponovno se duplicira stanje programa:

Ovim putem smo došli do retka ERROR

za (`x > y + 10`):

σ : $x \rightarrow x0$
$y \rightarrow y0$
$z \rightarrow 2*y0$

SPC : $x0 = 2*y0 \wedge x0 > y0 + 10$

za (`x ≤ y + 10`):

σ : $x \rightarrow x0$
$y \rightarrow y0$
$z \rightarrow 2*y0$

SPC : $x0 = 2*y0 \wedge x0 \leq y0 + 10$

Obrada ograničenja

- U svakom trenutku izvođenja programa, može se pokrenuti **rješavač ograničenja** (danas najčešće SMT-rješavač) koji će pronaći pridruživanje varijablama koje će biti model od SPC
- Pokretanje rješavača obično se provodi:
  - **na mjestima grananja**, kako bi se spriječilo daljnje istraživanje putova ako su ograničenja za taj put nezadovoljiva ili
  - **na mjestu gdje je došlo do pogreške**, kako bi se utvrdio **primjer konkretnih ulaznih vrijednosti** koji je doveo do pogreške
    - Npr. za (`x > y + 10`) došli smo do retka ERROR i razrješava se ograničenje `x0 = 2*y0` $\wedge$ `x0 > y0 + 10`
- Tako se pronalaze pogreške u složenim programima koje klasični postupci ispitivanja ne mogu jednostavno pronaći

KONKRETNO/SIMBOLIČKO IZVRŠAVANJE

Problem simboličkog izvršavanja

- **Ključni problem klasičnog simboličkog izvršavanja: ispitni slučaj ne može se generirati ako SMT-rješavač ne može (učinkovito) razriješiti ograničenje**

- Primjer: za nelinearna ograničenja koja se rješavaju u vidu T_{PA} (koja uključuje multiplikaciju), **nemamo garanciju** da će SMT-rješavač uspjeti pronaći model ili opovrgnuti SPC
  - Slično vrijedi i za neke kombinacije teorija, a problem je i komunikacija s vanjskim knjižnicama koje koriste funkcije za koje ne znamo kako se ponašaju

Rješenje za simboličko izvršavanje

- **Kombinirati klasično konkretno izvršavanje i simboličko izvršavanje, tzv. konkoličko izvršavanje (engl. *concolic execution, concolic testing*)**
- Danas se najčešće koristi pristup **dinamičkog simboličkog izvršavanja** (engl. *dynamic symbolic execution*, DSE)
  - Konkretno izvršavanje upravlja simboličkim izvršavanjem tako što se **program izvodi s konkretnim vrijednostima varijabli, ali se uz to pamte i simboličke vrijednosti** (program ima konkretno i simboličko stanje)
  - Nema garancije da će se istražiti svi putovi
- Postoje i alternative, npr. izvršavanjem generirano ispitivanje (engl. *execution generated testing*, EGT), selektivno simboličko izvršavanje (engl. *selective symbolic execution*, S^2E), itd.

DSE – primjer izvršavanja

Program:

```
int twice(int v) {
  return (v * v) % 50;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
        ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

- Izmijenjeni primjer sadrži nelinearno ograničenje

za (x == z):

σ : x → x0
y → y0
z → (y0*y0)%50

SPC : x0 = (y0*y0) %50

Problem!

Primjer DSE-izvršavanja „problematičnog" programa

Program:

```
int twice(int v) {
  return (v * v) % 50;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
        ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

- Funkcija `read()` dobavlja konkretne podatke, npr. x = 22, y = 7:

σ_k : x → 22, y → 7 — **SPC** = true (zasad još nema posjećenih grananja)

σ : x → x0, y → y0 — **SPC** = true (zasad još nema posjećenih grananja)

σ_k : x → 22, y → 7, z → 49 — **SPC** = true (zasad još nema posjećenih grananja)

σ : x → x0, y → y0, z → (y0*y0)%50 — **SPC** = true (zasad još nema posjećenih grananja)

- Ide se granom (`x ≠ z`):

σ_k : x → 22, y → 7, z → 49 — **SPC :** x0 ≠ (y0*y0)%50

σ : x → x0, y → y0, z → (y0*y0)%50 — **SPC :** x0 ≠ (y0*y0)%50

Primjer DSE-izvršavanja „problematičnog" programa

Program:

```
int twice(int v) {
  return (v * v) % 50;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
        ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

$\sigma_k : x \rightarrow 22$, $y \rightarrow 7$, $z \rightarrow 49$ **SPC :** $x0 \neq (y0*y0)\%50$

$\sigma : x \rightarrow x0$, $y \rightarrow y0$, $z \rightarrow (y0*y0)\%50$ **SPC :** $x0 \neq (y0*y0)\%50$

- Ovdje imamo nelinearno ograničenje `x0 ≠ (y0*y0)%50`. Ako želimo istražiti drugu granu, dakle za (`x == z`), negiramo ograničenje, no opet dobivamo nelinearno ograničenje `x0 = (y0*y0)%50`
- Ovdje DSE-izvršavanje pojednostavi ograničenje ubacivanjem konkretne vrijednosti y0 = 7, čime se rješavanjem SPC dobiva x0 = 49
- Zatim ponovno pokreće program s ulazom: `x → 49, y → 7`
- Sad zaista dolazimo do retka "ERROR"
- Međutim, ako iskoristimo DSE-izvršavanje, onda u ovom slučaju ne istražimo granu (`x ≤ y + 10`)

Problem eksplozije broja putova

- Kako bi se istražili i alternativni putovi, potrebno je svaki put **negirati SPC** do kojeg se došlo u ranijem istraživanju
- Istraživanje samo mogućih putova, pokazalo se:
  - Manje od 20% grananja imaju oba puta ostvariva*
  - Manje od 42% naredbi ovise o simboličkom *inputu**
- Heuristike pretraživanja putova (prostora stanja)
  - U dubinu – naprije se istražuju najdublja grananja
  - Slučajni put – na svakom grananju gdje su oba puta ostvariva, odaberi slučajno put koji treba dalje istražiti - začudo, pokazuje se izvrsnom heuristikom
  - Korištenje statičkog grafa kontrolnog toka (engl. *control flow graph*) programa, da se usmjeri istraživanje puta najbližeg dotad nepokrivenoj naredbi

* Cadar, C., Ganesh, V., Pawlowski, P., Dill, D. and Engler, D. EXE: Automatically generating inputs of death. In *Proceedings of CCS'06, (Oct–Nov 2006). An extended* version appeared in ACM TISSEC 12, 2 (2008).

Alati za simboličku verifikaciju

- DART (2005.+)
  - Izvorno konkoličko ispitivanje: slučajno ispitivanje + generiranje ispitnih slučajeva, provjera modela -> jezik C
- CUTE (*A Concolic Unit Testing Engine*) i jCUTE (*CUTE for Java*) (2006.+)
  - Nadogradnja DART-a za rad s višedretvenošću
  - http://osl.cs.illinois.edu/software/jcute/
- EXE i KLEE (2008.+)
  - Korite izvršavanjem generirano ispitivanje. veliki broj istovremenih stanja, ispitivanje programa u raznim programskim jezicima nakon pretvorbe u LLVM međujezik
  - https://klee.github.io/
- S^2E (2011.+)
  - Virtualni stroj sa selektivnim simboličkim izvršavanjem i modularnim analizatorima puta
  - http://s2e.epfl.ch/
- Java PathFinder, paket "symbolic" (2008.+):
  - https://github.com/SymbolicPathFinder/jpf-symbc

Materijali pripremljeni na temelju

1. L. de Moura, N. Bjørner, "Satisfiability Modulo Theories: Introduction and Applications," Communications of the ACM vol. 54, no. 9, pp.69-77, Sep. 2011
2. R. Piskač, Software Aided Verification, Lectures 8-15, Yale University, 2020
3. C. Cadar, K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," Communications of the ACM vol. 56, no. 2, pp.82-90, Feb. 2013.
4. R. Baldoni et al., "A Survey of Symbolic Execution Techniques", ACM Computing Surveys, vol. 51, issue 3, p. 50, pp 1–39, 2019.
5. M. Vechev, "Software Architecture and Engineering: Part II," Software Reliability Lab, ETH Zurich, 2015.
6. D. Paqué, "From Symbolic Execution to Concolic Testing," Institute of Theoretical Computer Science, Carl-Friedrich-Gauß-Fakultät ,Technische Universität Braunschweig, 2014.

FVPP: Hoareova logika i Dafny

Automatsko dokazivanje ispravnosti programa

Pripremio: izv. prof. dr. sc. Alan Jović

Ak. god. 2022./2023.

Sadržaj

- Dokazivanje ispravnosti programa
- Hoareova logika
  - Semantika programa
  - Jezik IMP
  - Jezik tvrdnji za Hoareovu logiku
  - Pravila prirodnog zaključivanja
  - Primjeri
- Programski alat Dafny

DOKAZIVANJE ISPRAVNOSTI PROGRAMA

Dokazivanje ispravnosti vs. provjera modela vs. ispitivanje

- **Provjera modela je jednostavnija od dokazivanja ispravnosti**
  - Provjeravaju se pojedinačna jednostavna svojstva programa
  - Ne iskazuju se potpuni zahtjevi (ugovori) o programu
  - Vrlo malo sustava se može dokazati direktno – zahtijeva se izgradnja modela
  - Dobro radi za kritični softver i hardver, slabije radi na podatkovno-intenzivnom i korisničkom softveru
- **Provjeru modela se lakše automatizira i sličnija je ispitivanju od dokazivanja ispravnosti**
  - Dokazivanje ispravnosti je mnogo temeljitije, teško se automatizira
- **Provjera modela pronalazi više nedostataka od ispitivanja**
  - Provjera modela prolazi **svim putovima** kroz **apstrahirani program**
  - Ispitivanje prolazi **nekim putovima** kroz **cjelokupni program**

Dokazivanje ispravnosti programa

- Za neki računalni program želimo **dokazati** da ispravno radi
  - Matematički dokaz na temelju poznavanja **koda programa** i postavljanja **uvjeta** na njegovo izvođenje u obliku **anotacija**
  - Primjer koda i uvjeta:

```
//: assume (x > 0)
def simple (Int x)
//: ensures y > 0
{
  val y = x - 2
  return y
}
```

Preduvjet

Uvjet po završetku programa (postuvjet)

Program

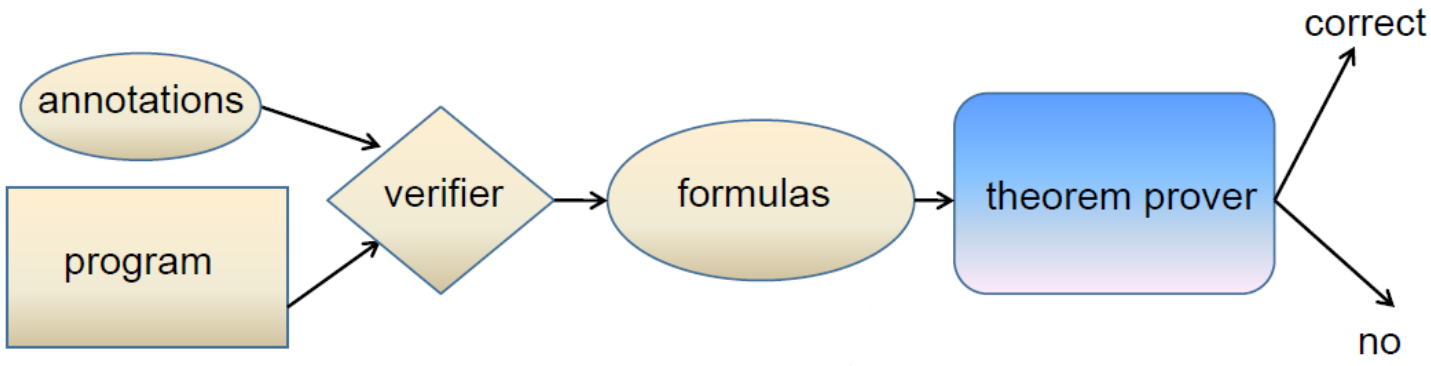
$$\forall x. \forall y. x > 0 \wedge y = x - 2 \rightarrow y > 0$$

Dokazivanje ispravnosti programa

- Anotacije piše programer ili analitičar programske potpore (engl. *software analyst*)
  - **Dodaju se izvornom kodu da bi izrazili svojstva koja se trebaju rasuditi o programu**
  - Tipični primjeri anotacija su one vezane uz **ugovor**:
    - **Preduvjeti** (engl. *precondition*) – opisuju nužna svojstva ulaznih vrijednosti u neki program
    - **Postuvjeti** (engl. *postcondition*) – opisuju što program treba napraviti (do čega treba dovesti)
    - **Invarijante** (engl. *invariants*) – opisuju svojstva koja trebaju vrijediti u svakom trenutku izvođenja programa
  - Ako anotacijsko svojstvo ne vrijedi, trebali bismo moći otkriti pogrešku u kodu programa

Dokazivanje ispravnosti programa

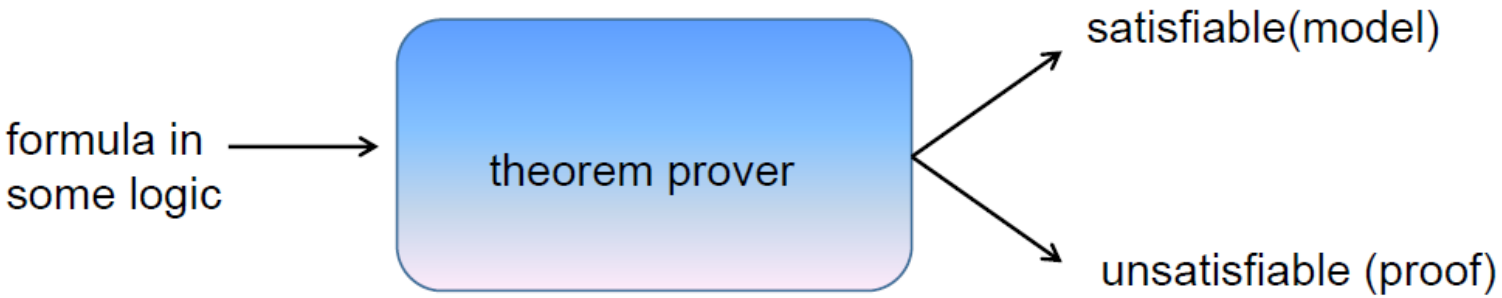
- Za automatsko dokazivanje ispravnosti programa razvijeni su programi za automatsko dokazivanje teorema (engl. *theorem prover*), ili jednostavnije „dokazivači" (engl. *prover*)



- Razlika u odnosu na dokazivanje formula u vremenskoj propozicijskoj logici: **stvarni programi imaju bitno složeniju matematičku teoriju** (logiku)

Dokazivanje ispravnosti programa

- **Procedura odlučivanja** (engl. *decision procedure*)
  - Algoritam koji odgovara na pitanje je li ulazna formula zadana u nekoj logici zadovoljiva ili ne
    - Npr. formula $x \leq y$ je zadovoljiva za $x = 0,\ y = 1$
    - Npr. formula $x \leq y$ i $x + 1 > y + 1$ nije zadovoljiva



  - U praksi, dokazivač nad stvarnim programima je vrlo složen algoritam koji se danas zasniva na **SAT-rješavačima** i **SMT-rješavačima** (tema ranijeg predavanja)

Dokazivanje ispravnosti programa

```
//: assume (x > 5)
def simple (Int x)
//: ensures y > 7
{
  val y = x + 2
  return y
}
```

- Što nam sve treba da dokažemo ispravnost?
  - Programski jezik
  - Izvođenje programa (i način za prikazati „kad god je program u nekom stanju...”)
  - Jezik za anotacije
  - Način da to izkombiniramo

- Želimo izraziti i dokazati: „uvijek kada program primi kao ulaznu vrijednost *x*, takvu da je *x* > 5 i pokrenemo program, rezultat *y* će biti takav da vrijedi *y* > 7”
- Želimo striktnu formalnu definiciju kada je program ispravan. Zašto?
  - Zato da možemo izgraditi **automatski alat** koji će znati kada je taj cilj postignut
- Problem: **kako formalno specificirati programski jezik?**

Formalna semantika jezika

Vrste formalne semantike programskih jezika:

- Denotacijska semantika (engl. *Denotational Semantics*)
  - Značenje programa je definirano konstrukcijom matematičkog objekta (tzv. denotacija) koji opisuju značenje onoga što program izražava (npr. djelomična funkcija – funkcija nije definirana za sve ulazne parametre, kao što je dijeljenje cijelih brojeva)
  - Primjer: **Apstraktna interpretacija** (engl. *Abstract Interpretation*)
- Aksiomatska semantika (engl. *Axiomatic Semantics*)
  - Značenje programa je definirano kao učinak koji program ima na istinitost logičkih tvrdnji
  - Primjer: **Hoareova logika** (engl. *Hoare Logic*)
- Operacijska semantika (engl. *Operational Semantics*)
  - Značenje programa je definirano formalizacijom pojedinačnih koraka izračunavanja u programu
  - Primjer: **Označeni tranzicijski sustavi** (engl. *Labeled Transition Systems*)

Dokazivanje ispravnosti programa

- Primjer programskog jezika: **Java**
- **Java Language Specification** (JLS) za Javu 8 (LINK) opisuje semantiku Java programa
  - Dokument ima 780 stranica (malo bolje za Python 3 – „samo” 135 stranica)
  - 148 stranica samo za definiciju semantike izraza
  - 48 stranica za definiciju semantike pozivanja metoda
- Semantika Jave (i drugih jezika) je definirana samo prozno (za razliku od sintakse)
- Kako možemo napraviti tu semantiku formalnom?
- Trebamo matematički **model** računanja u programima – netrivijalno za postići!

HOAREOVA LOGIKA

Hoareova logika

- Sir Charles Antony Richard (Tony) Hoare, r. 1934.
- **Hoareova logika** (1969.+) je formalni sustav koji koristi skup deduktivnih pravila zaključivanja za rasuđivanje nad računalnim programima
- Zasniva se na **Hoareovoj trojci** (tripletu):

  $\{P\}\ C\ \{Q\}$

  gdje su P i Q tvrdnje a C je naredba programa
- P se naziva **preduvjet** (engl. *precondition*), a Q **postuvjet** (engl. *postcondition*)
- Semantika Hoareove trojke:
  - Ako je **ispunjen preduvjet** pokreće se **naredba** koja, ako završi, dovodi do **ispunjavanja postuvjeta**

Značaj Hoareove logike

- Razvija se zadnjih 50 godina s težnjom automatske provjere ispravnosti računalnih programa
- Na nju se nadograđuju brojna proširenja za provjeru ispravnosti, npr.
  - Separacijska logika (engl. *separation logic*)
  - Združena logika (engl. *bunched logic*)
  - Hoareova logika za vremenska ograničenja (engl. *Hoare logic for time constraints*)
  - Relacijska Hoareova logika (engl. *relational Hoare logic*)
- Hoareova logika je korištena za dokazivanje ispravnosti većeg broja korisnih algoritama i programa, npr.
  - *Quicksort*
  - *Timsort*
  - Linearni programi (npr. za linearne diferencijalne jednadžbe i sl.)
  - Kvantni programi

Potpuna i djelomična ispravnost

- Hoareova logika ima precizniju **aksiomatsku semantiku** koja **zahtijeva djelomičnu ispravnost programa**
- **Djelomična ispravnost**
  - Ako program završi izvođenje počevši od ispunjenih određenih preduvjeta tada on ispunjava željene postuvjete. Nema garancije da se program treba završiti
- **Potpuna ispravnost** (nije zahtijevana Hoareovom logikom)
  - Program koji počinje od ispunjenih preduvjeta uvijek završava i ispunjava željene postuvjete
- Kako bismo formalizirali provjeru ispravnosti, trebamo:
  1. Objasniti formalnu semantiku,
  2. Dati primjer jezika (sintakse i semantike) na kojem možemo provjeriti ispravnost
  3. Navesti pravila zaključivanja u tom jeziku u kontekstu Hoareove logike

Aksiomatska semantika

- Aksiomatska semantika sastoji se od:
  - Jezika za navođenje **tvrdnji** (u praksi: anotacija) (engl. *assertions*) o programima
  - Pravila za ustanovljavanje istinitosti o tvrdnjama
- Neke tipične tvrdnje:
  - „Ovaj program završava s izvođenjem"
  - „Ako ovaj program završava s izvođenjem, varijable *x* i *y* imaju istu vrijednost tijekom izvršavanja programa"
  - „Pristup elementima polja je uvijek unutar ograničenja veličine polja"
- Tipični jezici za iskaz tvrdnji:
  - **Logika predikata prvoga reda**
  - Druge vrste logike (npr. vremenska), FOPL s proširenjima (teorijama)
  - Drugi specijalni specifikacijski jezici (npr. Z, JML)

IMP – jednostavni imperativni jezik

- Uvedimo primjer jezika na kojem možemo ustanoviti ispravnost
  - bitno pojednostavljeni imperativni programski jezik naziva „IMP”
- Sintaksa jezika IMP:
  - $n \in \mathbb{Z}$ – cijeli brojevi
  - *true*, *false* $\in$ **B** – Boolean
  - $x, y \in Vars$ – varijable programa
  - $e \in \boldsymbol{Aexp}$ – aritmetički izrazi
  - $b \in \boldsymbol{Bexp}$ – Booleovi izrazi
  - $c \in \boldsymbol{Com}$ – naredbe

IMP – jednostavni imperativni jezik

- Sintaksa aritmetičkih izraza ***Aexp***
  - $e ::= n,\ n \in \mathbb{Z}$

    $\mid x,\ x \in Vars$

    $\mid e_1 + e_2$

    $\mid e_1 - e_2$

    $\mid e_1 * e_2$

- Varijable se ne deklariraju prije korištenja
- Sve varijable su cjelobrojnog tipa
- Izrazi nemaju nikakve nuspojave (ne utječu na ostale varijable i izraze)

IMP – jednostavni imperativni jezik

- Sintaksa Booleovih izraza ***Bexp***
  - $b ::= true$

    $| \, false$

    $| \, e_1 = e_2$, za $e_1, e_2 \in Aexp$

    $| \, e_1 \leq e_2$, za $e_1, e_2 \in Aexp$

    $| \, \neg b \in Bexp$

    $| \, b_1 \wedge b_2$, za $b_1, b_2 \in Bexp$

    $| \, b_1 \vee b_2$, za $b_1, b_2 \in Bexp$

IMP – jednostavni imperativni jezik

- Sintaksa naredbi ***Com***
  - $c ::= skip$

    $\mid x = e$

    $\mid c_1; c_2$, za $c_1, c_2 \in Com$

    $\mid \text{if } b \text{ then } c_1 \text{ else } c_2$

    $\mid \text{while } b \text{ do } c$

- Napomena: jeziku nedostaju reference, funkcijski pozivi, druge vrste petlji, polja, složenije strukture podataka ...

IMP – jednostavni imperativni jezik

- Semantika jezika IMP
  - Značenje izraza e u IMP-u ovisi o vrijednostima varijabli, tj. o trenutačnom stanju
  - Stanje je u svakom trenutku predstavljeno funkcijom od Vars u $\mathbb{Z}$
  - Skup svih stanja je $Q = \text{Vars} \rightarrow \mathbb{Z}$
  - Koristimo malo q da izrazimo pojedinačna stanja iz Q

Semantika IMP-a: presude

- Pišemo <e, *q*> $\Downarrow$ *n* u značenju da se izraz e **evaluira** kao *n* u stanju *q*.
- Formula <e, *q*> $\Downarrow$ *n* naziva se **presuda**
  - Presuda je izjava o odnosu između e, *q* i *n*
- U ovom slučaju presuda se može razmatrati kao funkcija dvaju argumenata: e i *q* koja vraća *n*
- Ovakva formulacija naziva se **prirodna operacijska semantika** (engl. *natural operational semantics*)
  - Presuda dovodi u odnos izraz e i njegovo značenje *n*

Pravila zaključivanja

- Pravila evaluiranja (engl. *evaluation rules*) izraza izražavamo u obliku **pravila zaključivanja** (engl. *inference rules*) za naše presude
- Pravilo zaključivanja:

$$\frac{F_1 \ldots F_n}{G}, \text{ gdje } H$$

definira relaciju između presuda $F_1 \ldots F_n$ i G

- Presude $F_1 \ldots F_n$ su premise pravila zaključivanja
- Presuda G je zaključak pravila
- Formula H je sporedni uvjet (engl. *side condition*) pravila
- Ako je n=0 onda je pravilo aksiom i crta se može ispustiti

Pravila zaključivanja za IMP *Aexp*

- Uglavnom po jedno pravilo zaključivanja za svaki konstrukt jezika:

$<n, q> \Downarrow n$ - aksiom

$<x, q> \Downarrow q(x)$ - aksiom

$$\frac{<e_1,q>\Downarrow n_1 \quad <e_2,q>\Downarrow n_2}{<e_1+e_2,q>\Downarrow(n_1+n_2)}$$ - adicijsko pravilo

$$\frac{<e_1,q>\Downarrow n_1 \quad <e_2,q>\Downarrow n_2}{<e_1-e_2,q>\Downarrow(n_1-n_2)}$$ - suptrakcijsko pravilo

$$\frac{<e_1,q>\Downarrow n_1 \quad <e_2,q>\Downarrow n_2}{<e_1*e_2,q>\Downarrow(n_1*n_2)}$$ - multiplikacijsko pravilo

- Pravila su definirana na osnovi strukture izraza – **strukturalna operacijska semantika**

Pravila zaključivanja za IMP *Bexp*

$<true, q> \Downarrow true$ - aksiom

$<false, q> \Downarrow false$ - aksiom

$$\frac{<e_1,q>\Downarrow n_1 \quad <e_2,q>\Downarrow n_2}{<e_1=e_2,q>\Downarrow(n_1=n_2)}$$

$$\frac{<e_1,q>\Downarrow n_1 \quad <e_2,q>\Downarrow n_2}{<e_1\leq e_2,q>\Downarrow(n_1\leq n_2)}$$

$$\frac{<b_1,q>\Downarrow t_1 \quad <b_2,q>\Downarrow t_2}{<b_1\, i\, b_2,q>\Downarrow(t_1\, i\, t_2)}, i = \wedge$$

Kako čitati pravila zaključivanja

- **Unaprijed**
  - Ako znamo da presude u premisama vrijede onda možemo zaključiti i da presuda u zaključku vrijedi, npr.

$$\frac{<2,q>\Downarrow 2 \qquad <3,q>\Downarrow 3}{<2*3,q>\Downarrow 6}$$

- **Unazad, rasuđivanjem inverzijom**
  - Npr. ako želimo evaluirati $e_1 + e_2$ uočavamo da zaključak $e_1 + e_2 \Downarrow n$ mora biti **adicijsko** pravilo (samo je jedno takvo)
  - Rekurzivno idemo unatrag po pravilima, u svakom koraku primijenimo jedno pravilo dok ne dođemo do najosnovnije evaluacije (u ovom slučaju određivanje n_1 i n_2)

Kako čitati pravila zaključivanja

- 1. korak:

$$\frac{<\mathrm{x}, \{x \mapsto 3, y \mapsto 2\}> \Downarrow 3 \qquad <2 * \mathrm{y}, \{x \mapsto 3, y \mapsto 2\}> \Downarrow 4}{<\mathrm{x} + (2 * \mathrm{y}), \{x \mapsto 3, y \mapsto 2\}> \Downarrow 7}$$

- 2. korak:

$$\frac{<\mathrm{x}, \{x \mapsto 3, y \mapsto 2\}> \Downarrow 3 \qquad \frac{<y, \{x \mapsto 3, y \mapsto 2\}> \Downarrow 2 \quad <2, \{x \mapsto 3, y \mapsto 2\}> \Downarrow 2}{<2 * \mathrm{y}, \{x \mapsto 3, y \mapsto 2\}> \Downarrow 4}}{<\mathrm{x} + (2 * \mathrm{y}), \{x \mapsto 3, y \mapsto 2\}> \Downarrow 7}$$

- Tu stajemo, jer smo došli do aksioma: $<2, q> \Downarrow 2$ i $<y, q> \Downarrow 2$

Semantika IMP-a: naredbe

- Evaluacija naredbe (*Com*) ima nuspojave, ali nijedan izravni rezultat
- Rezultat naredbe *c* pokrenute u predstanju *q* je **prijelaz** iz *q* u poststanje *q*' :

$$q \xrightarrow{c} q'$$

- Formalizacija evaluacije naredbi može se napraviti pomoću **označenih tranzicijskih sustava** (engl. *labeled transition systems*)

Označeni tranzicijski sustav

- Označeni tranzicijski sustav (kraće: LTS) je struktura

  $LTS = (Q, Act, \rightarrow)$ gdje je:

  Q – skup **stanja** sustava

  Act – skup **akcija**

  $\rightarrow \subseteq Q \times Act \times Q$ – relacija prijelaza (engl. *transition relation*)

- Pišemo $q \xrightarrow{a} q'$ za $(q, a, q') \in \rightarrow$

- Potrebno je definirati semantiku za sve naredbe (akcije) a LTS-a za jezik IMP, pri čemu razmatramo promjene stanja q kao rezultat djelovanja naredbi

Semantika IMP-a: naredbe putem LTS

- $q \xrightarrow{skip} q$ (za skip)

$$\frac{q \xrightarrow{c_1} q' \quad q' \xrightarrow{c_2} q''}{q \xrightarrow{c_1\ ;\ c_2} q''}$$ (za $c_1\ ; c_2$)

- $$\frac{<e,q> \Downarrow n}{q \xrightarrow{x:=e} q\ ++\{x \mapsto n\}}$$ (za x = e)

- $$\frac{<b,q> \Downarrow true \quad q \xrightarrow{c_1} q'}{q \xrightarrow{\text{if } b \text{ then } c_1 \text{ else } c_2} q'}$$ $$\frac{<b,q> \Downarrow false \quad q \xrightarrow{c_2} q'}{q \xrightarrow{\text{if } b \text{ then } c_1 \text{ else } c_2} q'}$$ (za if b then c_1 else c_2)

- $$\frac{<b,q> \Downarrow false}{q \xrightarrow{while\ b\ do\ c} q}$$ $$\frac{<b,q> \Downarrow true \quad q \xrightarrow{c} q' \quad q' \xrightarrow{while\ b\ do\ c} q''}{q \xrightarrow{while\ b\ do\ c} q''}$$ (za while b do c)

Semantika IMP-a: prijelazi

- Tvrdnje za jezik IMP mogu se iskazati u obliku
- {A} *c* {B} sa značenjem da:
- ako A vrijedi u stanju q i $q \xrightarrow{c} q'$ onda B vrijedi u q'
- A je preduvjet, a B je postuvjet
- Primjer tvrdnje:

  $\{ y \leq x \}\ z := x;\ z := z + 1\ \{ y < z \}$
- Ova tvrdnja je validna (vrijedi)

- Tvrdnje ovog tipa za jezik IMP su Hoareove trojke (tripleti) koje su djelomično ispravne

Jezik tvrdnji za Hoareovu logiku

- Za iskaz tvrdnje u Hoareovoj logici koristi se **predikatna logika prvoga reda** zajedno s izrazima u **IMP-u**:
- A :: = *true* | *false* | $e_1 = e_2$ | $e_1 \geq e_2$ | $A_1 \wedge A_2$ | $A_1 \vee A_2$ | $A_1 \Rightarrow A_2$ | $\forall x.A$ | $\exists x.A$
- Implicitno, sve IMP varijable imaju raspon cijelih brojeva
- Svi IMP-ovi Booleovi izrazi su isto tako tvrdnje A.

- Značenje aksiomatskog jezika:
  - Oznaka $q \vDash A$ znači da tvrdnja A vrijedi u danom stanju q.
  - Presuda $\vDash$ je definirana induktivno na strukturi tvrdnji
  - Oznaka $\vDash A$ znači da tvrdnja A vrijedi u bilo kojem stanju, uvijek je istinita

Semantika tvrdnji za Hoareovu logiku

- $q \vDash true$ Stanje q uvijek vrijedi
- $q \vDash e_1 = e_2$ ako i samo ako $<e_1,q>\Downarrow = <e_2,q>\Downarrow$
- $q \vDash e_1 \geq e_2$ ako i samo ako $<e_1,q>\Downarrow \geq <e_2,q>\Downarrow$
- $q \vDash A_1 \wedge A_2$ ako i samo ako $q \vDash A_1$ i $\mathrm{q} \vDash A_2$
- $q \vDash A_1 \vee A_2$ ako i samo ako $q \vDash A_1$ ili $q \vDash A_2$
- $q \vDash A_1 \Rightarrow A_2$ ako i samo ako $q \vDash A_1$ implicira $q \vDash A_2$
- $q \vDash \forall x.\mathrm{A}$ ako i samo ako $\forall n \in \mathbb{Z}.\ q[x{:=}n] \vDash \mathrm{A}$
- $q \vDash \exists x.\mathrm{A}$ ako i samo ako $\exists n \in \mathbb{Z}.\ q[x{:=}n] \vDash \mathrm{A}$

Formalna definicija djelomične i potpune ispravnosti

- Djelomična ispravnost:
- $\vDash \{A\}\ c\ \{B\}$ ako i samo ako $\forall q \in Q.\ \forall q' \in Q.\ q \vDash A \wedge q \xrightarrow{c} q' \Rightarrow q' \vDash B$

- Potpuna ispravnost:
- $\vDash [A]\ c\ [B]$ ako i samo ako $\forall q \in Q.\ q \vDash A \Rightarrow \exists q' \in Q.\ q \xrightarrow{c} q' \wedge q' \vDash B$

- q je stanje, definira vrijednosti varijabli
- {A} c {B} – Hoareova trojka
- $q \vDash F$ – u stanju q vrijedi formula F

- **Važan rezultat:** sada se može formalno dokazati da je program ispravan u smislu operacijske semantike
- **Problem:** program se mora pokrenuti da bi se dokazala njegova ispravnost – treba nam automatsko dokazivanje teorema bez pokretanja programa

Pravila prirodnog zaključivanja za propozicijsku logiku

- U propozicijskoj logici vrijede pravila prirodnog zaključivanja koja smo uveli na 2. predavanju
- Ona se mogu pisati s oznakom $\vdash A$ kada je A moguće zaključiti iz inicijalnih aksioma (A je dedukcija)
- Neka od pravila su (vidjeti 2. predavanje, zadnji slajd, za njih sve):

$$\frac{\vdash A \qquad \vdash B}{\vdash A \text{ i } B}$$ (uvođenje konjunkcije)

$$\frac{\vdash A}{\vdash A \text{ ili } B}$$ *ili* = $\vee$ (uvođenje disjunkcije)

$$\frac{\vdash A \Rightarrow B \qquad \vdash A}{\vdash B}$$ (modus ponens)

...

Pravila prirodnog zaključivanja za predikatnu logiku

- U predikatnoj logici prvog reda vrijede i dodatna pravila zaključivanja:

$$\frac{\vdash A[a/x]}{\vdash \forall x.\, A}$$ uvođenje univerzalnog kvantifikatora

$$\frac{\vdash \forall x.\, A}{\vdash A[e/x]}$$ eliminacija univerzalnog kvantifikatora

$$\frac{\vdash A[e/x]}{\vdash \exists x.\, A}$$ uvođenje egzistencijskog kvantifikatora

$$\frac{\vdash \exists x.\, A \qquad \begin{array}{c}\vdash A[a/x] \\ \vdots \\ \vdash B\end{array}}{\vdash B}$$ eliminacija egzistencijskog kvantifikatora

- pri čemu $A[a/x]$ označava operaciju supstitucije (unifikacije) – zamjenu svih slobodnih pojava varijable *x* u A s vrijednosti *a*, uz dodatni uvjet da je ***a* svježa vrijednost** (ne pojavljuje se nigdje drugdje)
- $A[e/x]$ označava zamjenu svih slobodnih pojava varijable *x* u A s **izrazom** e

Pravila prirodnog zaključivanja za Hoareovu logiku

- Po jedno pravilo za svaki sintaksni konstrukt jezika IMP (uzimajući u obzir predikatnu logiku prvoga reda:

- $\vdash \{A\}\ \text{skip}\ \{A\}$ – **aksiom skip** – prazna naredba ne mijenja stanje programa
- $\vdash \{A[e/x]\}\ x{:=}e\ \{A\}$ – **aksiom pridruživanja:** svaka slobodna pojava varijable *x* zamijenjena je izrazom e
- $\dfrac{\vdash \{A\}\ c_1\ \{B\} \qquad \vdash \{B\}\ c_2\ \{C\}}{\vdash \{A\}\ c_1\ ;\ c_2\ \{C\}}$ – **pravilo kompozicije**
- $\dfrac{\vdash \{A \wedge b\}\ c_1\ \{B\} \qquad \vdash \{A \wedge \neg b\}\ c_2\ \{B\}}{\vdash \{A\}\ \text{if b then}\ c_1\ \text{else}\ c_2\ \{B\}}$ – **pravilo uvjeta**
- $\dfrac{\vdash \{I \wedge b\}\ c\ \{I\}}{\vdash \{I\}\ \text{while}\ b\ \text{do}\ c\ \{I \wedge \neg b\}}$ – **pravilo petlje while**

- I dodatno **pravilo posljedice** (engl. *rule of consequence*):
- $\dfrac{\vdash A' \Rightarrow A \quad \vdash \{A\}\ c\ \{B\} \quad \vdash B \Rightarrow B'}{\vdash \{A'\}\ c\ \{B'\}}$

Invarijanta petlje

- *I* – invarijanta petlje je formula koja:
  - vrijedi **prije ulaza u petlju prvi put** i u **svim stanjima koja zadovoljavaju preduvjet**
  - njezina vrijednost se **održava (vrijedi) nakon svake iteracije petlje i na kraju izvođenja petlje**

Primjer: jednostavna petlja

$$\frac{\vdash \{\mathtt{I} \wedge b\}\ c\ \{\mathtt{I}\}}{\vdash \{\mathtt{I}\}\ \mathtt{while}\ b\ \mathtt{do}\ c\ \{\mathtt{I} \wedge \neg b\}}$$

- Želimo zaključiti da vrijedi:

  $\vdash \{x \leq 0\}\ \mathtt{while}\ \ x \leq 5\ \mathtt{do}\ \ x := x + 1\ \{x = 6\}$

- Trebamo iskoristiti pravilo za while s invarijantom petlje $I \equiv x \leq 6$

$$\frac{\vdash\{x \leq 6 \wedge x \leq 5\}\ x := x + 1\ \{x \leq 6\}}{\vdash\{\mathrm{x} \leq 6\}\ \mathtt{while}\ \ \mathrm{x} \leq 5\ \mathtt{do}\ \ \mathrm{x} := \mathrm{x} + 1\ \{\ \mathrm{x} \leq 6 \wedge \mathrm{x} > 5\}}$$

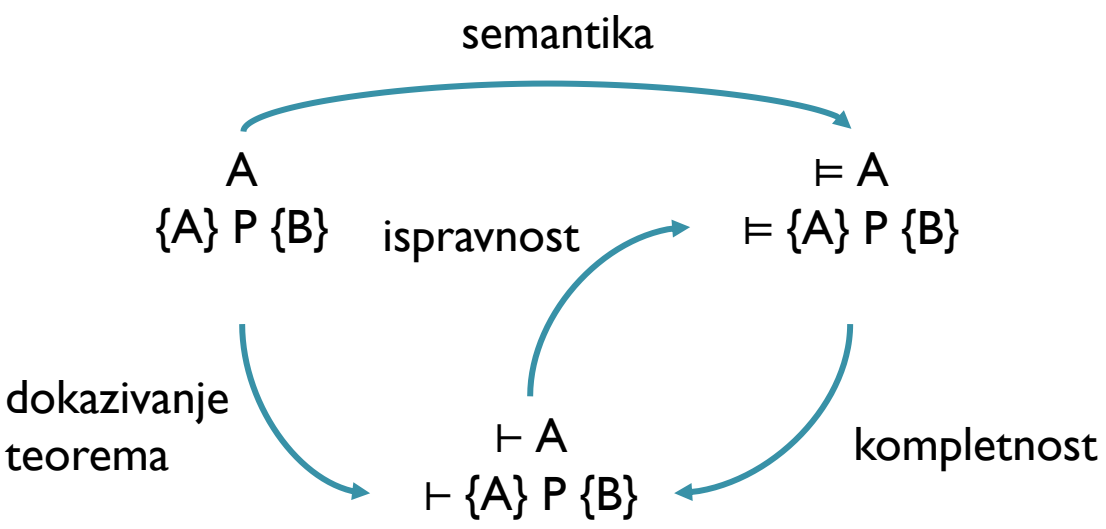
$$\frac{\vdash A' \Rightarrow A \quad \vdash\{A\}\ c\ \{B\} \quad \vdash B \Rightarrow B'}{\vdash\{A'\}\ c\ \{B'\}}$$

- Potom treba primijeniti pravilo posljedice da bismo dokazali tvrdnju:

$$\frac{\vdash x \leq 0 \Rightarrow x \leq 6 \qquad \vdash x \leq 6 \wedge x > 5 \Rightarrow x = 6 \qquad \vdash\{x \leq 6\}\ \mathtt{while}\ \ldots\ \{x \leq 6 \wedge x > 5\}}{\vdash\{\mathrm{x} \leq 0\}\ \mathtt{while}\ldots\ \{\mathrm{x} = 6\}}$$

Zaključno o Hoareovoj logici

- Imamo jezik za izražavanje tvrdnji o programima
- Znamo kada je takva tvrdnja istinita
- Imamo i simboličku metodu (pravila prirodnog zaključivanja) kako bismo izveli tvrdnje



Zaključno o Hoareovoj logici

- Hoareova logika omogućuje rasuđivati o programima
- Primjena pravila prirodnog zaključivanja u Hoareovoj logici nije jednostavna, ali se može automatizirati; pritom su problemi (ipak riješivi):
  - Kada primijeniti pravilo posljedice?
  - Koje invarijante koristiti za while petlju?
  - Kako dokazati implikacije uključene u pravilo posljedice?
- Ako je IMP program donekle složeniji dokaz može biti **dug**
- Što ako imamo potrebe za rasuđivanje o još složenijim programima nego što nam dozvoljava sintaksa i semantika IMP-a?
  - Trebaju nam teorije za zaključivanje sa složenijim strukturama podataka (polja, skupovi, liste)
  - Trebaju nam teorije za zaključivanje o zauzimanju memorije
- Ovo izlazi izvan okvira klasične Hoareove logike

PROGRAMSKI ALAT DAFNY

Programski alat Dafny

- Programski alat Dafny (2013.+) sastoji se od:
  - **imperativnog objektnog jezika za anotirane programe** i
  - **verifikatora koji prevodi anotirane programe u međujezik za platformu .NET**
- https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/
- https://dafny.org/
- Prevoditelj **statički** (statički verifikator) provjerava:
  - odsustvo pogrešaka pri izvođenju (engl. *runtime errors*)
  - mogućnost završetka petlji i poziva metoda
  - ispravnost korisnički definiranih ugovora

- Ref.: Rustan Leino, Michal Moskal, Co-Induction Simply: Automatic Co-Inductive Proofs in a Program Verifier, MSR-TR-2013-49 | July 2013

Programski alat Dafny

- Kao objektni jezik, Dafny podržava:
  - generičke razrede
  - reference na objekte, dinamičku alokaciju memorije
- Jezik podržava specifikaciju i verifikaciju:
  - **ugovore metoda** – preduvjete, postuvjete, uvjete okvira (engl. *frame conditions*) – klauzule koje izražavaju one objekte koji se smiju mijenjati
  - **invarijante petlji** i *inline* tvrdnje (unutar koda)
  - nepromijenjive tipove podataka (engl. *immutable types*): skupove, sekvence, algebarske tipove podataka
  - i dr.

Alat Boogie

- Alat **Boogie** (2008.+, Microsoft) je verifikator međujezika koji generira Dafny
- Boogie je potpuno automatizirani deduktivni verifikacijski sustav koji generira uvjete za verifikaciju koji se šalju **SMT-rješavaču** na provjeru (defaultni je **Z3**)
- SMT-rješavači znaju rasuđivati o složenijim teorijama nego što omogućuje jezik IMP i Hoareova logika
- Boogie koristi **semantiku najslabijeg preduvjeta**

Algoritam za verifikaciju (generiranje uvjeta verifikacije)

- Ideja: propagirati postuvjet **unazad** kroz program kako bismo došli do preduvjeta koji treba biti ispunjen – generiranje uvjeta za uspješnu verifikaciju
- Od $\{A\}$ P $\{B\}$:

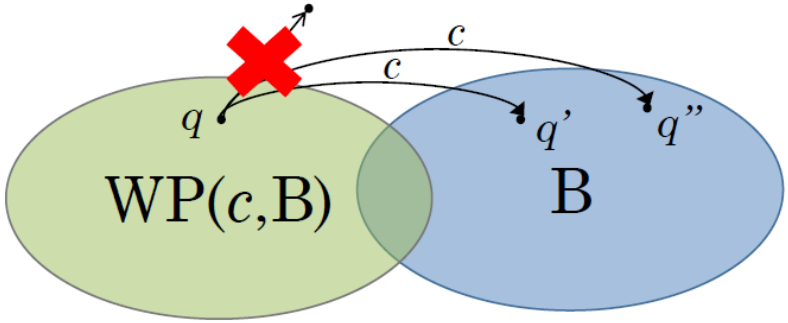
  generira se formula $A \Rightarrow F(P, B)$, gdje je $F(P, B)$ formula koja opisuje

  početna stanja za program koji će završiti u B
- Generiranje uvjeta za verifikaciju (engl. *Verification Condition Generation*, **VCG**)

- Propagacija unazad $F(P, B)$ može se formalizirati putem **najslabijih preduvjeta** (engl. *weakest precondition*)

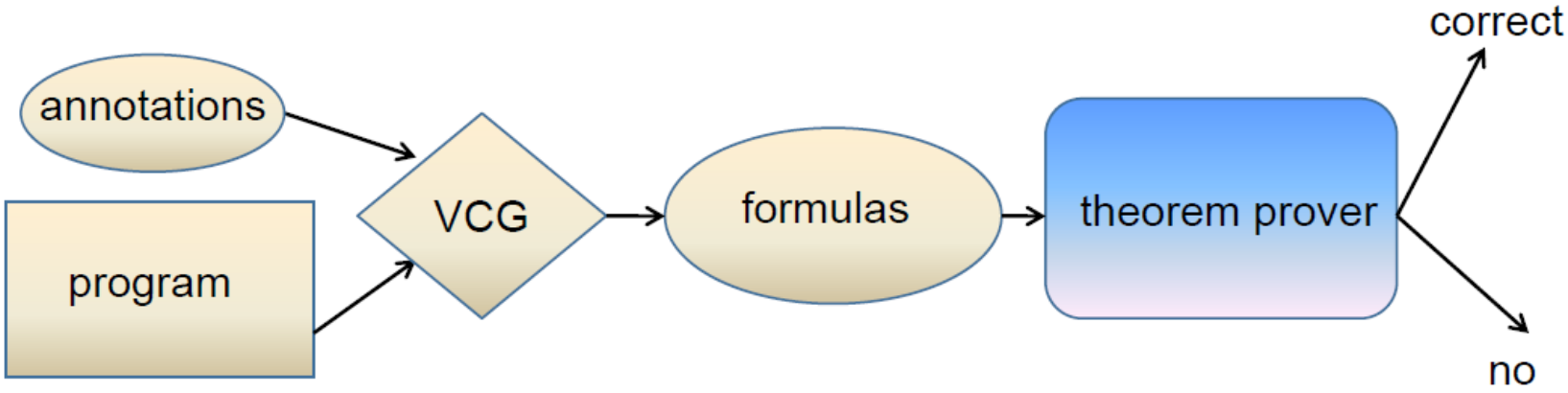
Semantika najslabijeg preduvjeta

- Dijkstra je predložio pristup implementaciji verifikacije programa koji se zove semantika najslabijeg preduvjeta
- **Najslabiji preduvjet WP(*c*, B) vrijedi za svako stanje *q* čija stanja c-sljedbenika sva zadovoljavaju B:**
- $q \vDash \mathrm{WP}(c,\mathrm{B})$ ako i samo ako $\forall q' \in Q.\ q \xrightarrow{c} q' \Rightarrow q' \vDash \mathrm{B}$



- Izračuna se WP(*P*,B) rekurzivno od kraja prema strukturi programa *P*.
- Ovaj pristup se razlikuje od Hoareovog izvornog pristupa budući da se inzistira na **potpunoj ispravnosti** umjesto na djelomičnoj ispravnosti uz **uvjet da je na početku samo zadan postuvjet**

Primjena SMT-rješavača



- VCG generira sveukupnu formulu ograničenja (koja se može razmatrati kao više pojedinačnih formula), a koja predstavlja naš program (neku konkretnu funkciju) za koji se treba utvrditi ispravnost
- Ovdje je *theorem prover* SMT-rješavač koji utvrđuje je li navedena formula ispravna

Dafny – deklaracija metode

- Primjer **metode**:

```
method Abs(x: int) returns (y: int)
{
  if (x < 0) { return-x; }
  else { y := x; }
}
```

- Nužno se eksplicitno navodi **povratna vrijednost (returns)** metode

Dafny – funkcije i predikati

- **Funkcije** su metode koje:
  - **ne smiju imati nuspojave** (sve što se događa, događa se samo unutar funkcije)
  - **uvijek moraju završiti** (potpuna ispravnost)
- Oba svojstva provjerava verifikator
- Mogu se koristiti unutar specifikacije
- **Predikati** su one funkcije koje vraćaju tip podatka bool

Dafny – preduvjeti i postuvjeti

- Preduvjeti i postuvjeti deklariraju se pomoću ključnih riječi requires i ensures
- Primjer:

```
method MultipleReturns (x: int, y: int)
returns(more: int, less: int)
  requires 0 < y
  ensures less < x < more
{
  more := x + y;
  less := x -y;
}
```

Dafny – rad s petljama

- Dafny ne zna za vrijeme prevođenja koliko puta će se petlja while izvesti
- Ipak, verifikator treba razmotriti **sve moguće putove** kroz program
- **Invarijante petlji služe da bi verifikator eliminirao sve petlje u programu koristeći indukciju**
- Invarijantu petlji nužno navodi programer / analitičar programa
- Podsjetimo: invarijanta petlji je **formula** koja:
  - **vrijedi prije ulaza u petlju prvi put**
  - **njezina vrijednost se održava (vrijedi) nakon svake iteracije petlje**

Dafny – rad s petljama – primjer

```
method computeFib(n: nat) returns (m: nat)
  ensures m == Fibonacci(n)
{
  var i:= 0;
  var k := 1;
  m := 0;
  while(i < n)
    invariant 0 <= i <= n
    invariant k == Fibonacci(i+1) && m == Fibonacci(i)
  {
    m, k := k, m + k;
    i := i + 1;
  }
}
```

```
function Fibonacci(n: nat): nat
  decreases n
{
  if n < 2 then n else Fibonacci(n-2) + Fibonacci(n-1)
}
```

Dafny – uvjeti okvira

- Funkcije i metode trebaju specificirati **memorijski otisak** – lokacije kojima mogu pristupiti i/ili mijenjati
- Skup memorijskih lokacija naziva se **okvir** (engl. *frame*)
- Uvjeti okvira:
  - **reads** S – specificira da **funkcija** čita samo lokacije u okviru S
  - **modifies** S – specificira da **metoda** mijenja samo lokacije u okviru S
- Funkcije smiju čitati samo one lokacije koje su specificirane s njihovim reads klauzulama (anotacijama)
- Metode smiju pristupiti bilo kojoj lokaciji ali smiju mijenjati samo one lokacije koje specificiraju njihove modifies klauzule

Dafny – uvjeti okvira – primjer

```
predicate sorted(a: array<int>)
  requires a != null
  reads a
{
  forall j, k :: 0 <= j < k < a.Length==> a[j] <= a[k]
}

method BinarySearch(a: array<int>, key: int)
returns(index: int)
  requires a != null && sorted(a)
  ensures...
{
  ...
}
```

Zaključak

- Prikazane su osnove dokazivanja ispravnosti programa korištenjem pravila zaključivanja u Hoareovoj logici
- Programski alat Dafny implementira zaključivanje koje se jednim dijelom temelji na Hoareovoj logici
  - značajna nadogradnja Hoareove logike i korištenje SMT-rješavača za automatsko zaključivanje
- Preporuke za daljnje učenje:
  - proučiti detaljnije kako je implementiran **Dafny**
  - proučiti **separacijsku logiku** i njezinu primjenu za verifikaciju programa sa zahtjevnijim memorijskim strukturama
  - proučiti alternativne notacije za specifikacije, npr. **JML**